TURING 图灵程序设计丛书 程序员修炼系列

WILEY

AntiPatterns
Refactoring Software, Architectures, and Projects in Crisis

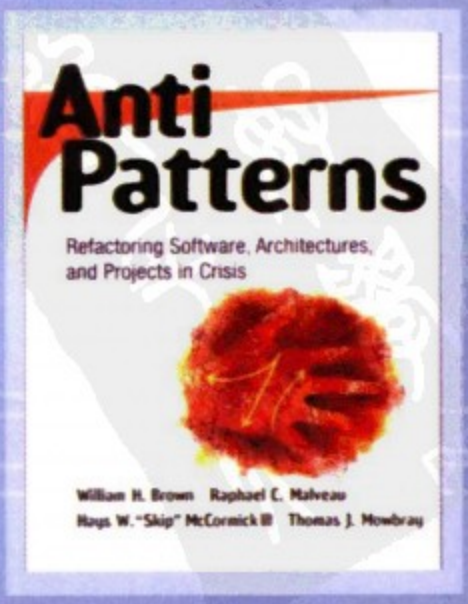
# 反模式

## 危机中软件、架构和项目的重构


第9届Jolt生产效率大奖

[美] William J. Brown
Raphael C. Malveau
Hays W. McCormick III
Thomas J. Mowbray 著

宋锐 等译




- 软件工程圣经之一
- 涵盖软件项目中开发、架构和管理各个方面
- 从独特的视角审视软件开发

人民邮电出版社
POSTS & TELECOM PRESS

AntiPatterns
Refactoring Software, Architectures, and Projects in Crisis

# 反模式 危机中软件、架构和项目的重构

“本书的作者们在软件开发的管理领域显然身经百战。我自己曾经见证了太多陷入困境的开发项目，因此书中一个又一个真知灼见都能引起我的强烈共鸣。书中字里行间都充分体现出作者们的丰富经验。”

——John Vlissides，软件开发大师，《设计模式》作者之一

“无论是程序员、架构师，还是技术经理，只要你从事软件开发工作，就应该熟悉本书中的内容——不仅要了解问题所在，还要掌握可能的解决之道。”

——Francis Glassborow，ACCU（C和C++用户协会）前主席，曾任ISO C和C++标准委员会主席

**本书与《设计模式》、《重构》、《解析极限编程》等巨著一起被誉为“软件工程四大圣经”，曾荣获*Software Development*杂志颁发的第9届Jolt生产效率大奖，对软件开发产生了广泛而深远的影响。如今，书中记录的The Blob、Spaghetti Code、Golden Hammer、Analysis Paralysis等反模式早已经融入广大开发社区，成为大家的日常用语。**

**从某种意义上讲，本书甚至比其他经典更具雄心，涵盖也更广。它深刻地揭示并试图解决这样一个现实问题：软件实现本身并非容易出错和出现疏忽的唯一所在，糟糕的软件架构设计和管理有时候更容易使软件开发陷入困境。本书不仅讨论软件的问题，还剖析软件架构和管理方面的常见错误，并将这些错误分门别类，对应给出凝聚了业界宝贵经验和惨痛教训的解决方案。通过本书，你将能够以史为鉴，避免重蹈覆辙。**

## 作者介绍

本书的四位作者都是拥有丰富软件开发、管理和培训经验的资深技术专家，他们创立并发展了反模式理论。

William J. Brown曾任Saga软件公司研发总监和OMG金融业工作组主席。擅长金融行业大型软件系统的开发。

Raphael C. Malveau是Eidea实验室的首席科学家，专注于企业业务对象框架的开发。

Hays W. McCormick III曾任著名软件企业Mitre公司的首席软件架构师，主持开发了许多大型分布式系统。

Thomas J. Mowbray曾任Blueprint公司首席科学家，OMG荣誉会士，国际组件管理组织创始人和主席。

WILEY
www.wiley.com

本书相关信息请访问：图灵网站 http://www.turingbook.com
读者/作者热线：(010)88593802
反馈/投稿/推荐信箱：

**分类建议** 计算机/程序设计

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-16279-3

9 787115 162793 >

ISBN 978-7-115-16279-3/TP

定价：45.00 元

TURING 图灵程序设计丛书 程序员修炼系列

AntiPatterns
Refactoring Software, Architectures, and Projects in Crisis

# 反模式
## 危机中软件、架构和项目的重构

[美] William J. Brown
Raphael C. Malveau
Hays W. McCormick III
Thomas J. Mowbray 著

宋锐 等译

人民邮电出版社
北 京

**图书在版编目（CIP）数据**

反模式：危机中软件、架构和项目的重构 /（美）布朗（Brown, W.J.）等著；宋锐等译. —北京：人民邮电出版社，2008.1
（图灵程序设计丛书）
ISBN 978-7-115-16279-3

Ⅰ. 反… Ⅱ. ①布…②宋… Ⅲ. 软件开发 Ⅳ. TP311.52

中国版本图书馆 CIP 数据核字（2007）第 074692 号

## 内 容 提 要

模式是可以复用的优秀解决方案。本书从一个新的角度审视模式，提出了反模式的概念，介绍了在软件开发中常常出现的问题——将设计模式错误应用于不适当的上下文环境。首先，定义了软件开发参考模型和文档模板来说明这些反模式。然后，从开发人员角度、架构角度和管理角度三个方面对这些反模式逐一说明，并说明了与特定反模式相关的背景、原因、症状和后果，让读者可以迅速地检验身边的项目是否出现了这些状况，同时也针对每个反模式给出了相应的解决方案。

本书适用于从事项目管理和软件开发的相关人员。

图灵程序设计丛书

**反模式：危机中软件、架构和项目的重构**

◆ 著 [美] William J. Brown Raphael C. Malveau
Hays W. McCormick III Thomas J. Mowbray
译 宋 锐 等
责任编辑 陈兴璐

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京顺义振华印刷厂印刷
新华书店总店北京发行所经销

◆ 开本：800×1000 1/16
印张：14.75
字数：363 千字 2008 年 1 月第 1 版
印数：1 – 5 000 册 2008 年 1 月北京第 1 次印刷
著作权合同登记号 图字：01-2007-2465 号

ISBN 978-7-115-16279-3/TP

定价：45.00 元

**读者服务热线：(010)88593802 印装质量热线：(010)67129223**

**反盗版热线：**（010）67171154

# 对本书的赞誉

“本书的作者们在软件开发的管理领域显然身经百战。我自己曾经见证了太多陷入困境的开发项目，因此书中一个又一个真知灼见都能引起我的强烈共鸣。书中字里行间都充分体现出作者们的丰富经验。”

——John Vlissides，软件开发大师，《设计模式》作者之一

“本书可以称得上我们行业的圣经，将使你能更全面地理解面向对象的程序设计。我从本书中受益匪浅。”

——Thomas E. Davis，Internet计划的首席技术官

“本书深入探讨了软件项目中的问题及其解决方案。它没有局限于技术模式，而是涵盖了软件开发的方方面面，包括管理和组织问题。它为我们行业创立了一个标准词汇表，大大方便了开发人员、架构师和经理之间的交流。”

——Angelika Langer，著名技术专家，*Agile Java*一书作者

“只要你从事软件开发工作，无论是程序员、架构师，还是技术经理，你和你的同事就都应该熟悉本书中的内容——不仅要了解问题所在，还要掌握可能的解决之道。”

——Francis Glassborow，ACCU（C和C++用户协会）前主席，
曾任ISO C和C++标准委员会主席

“软件管理人员、架构师和开发人员都可以从本书中吸取前人的惨痛教训。书中关于软件架构性的反模式对软件工程的贡献极大。”

——Kyle Brown，IBM资深模式和Java技术专家，
Enterprise Java Programming for IBM WebSphere一书作者

“本书秉承了《设计模式》一书开创的传统。作者们揭示并命名了反模式——糟糕的管理或者架构所导致的许多常见问题，也是大多数有经验的软件从业人员能够识别的错误。对于各种反模式，作者们还提供了解决之道。”

——Gerard Meszaros，ClearStream咨询公司首席科学家，Mock Object模式发明者

“我是一口气读完本书草稿的，其间不断被书中的深刻见解打动，而作者的幽默表述又常常使我会心一笑。作者们叙述的许多反模式都可以在我自己的经历中得到验证。强烈推荐。”

——Al Stevens，Dr. Dobb's Journal

“我喜欢本书，读起来非常有意思。我把书借给了一位英雄所见略同的项目经理，他居然不还我了……”

——Jeff Grigg，资深软件顾问

# 译 者 序

从1994年以来，有关设计模式的文献出版量呈指数形式增长。对有经验的面向对象架构师来说，现在有许多的、而且还在不断增长的可复用设计，可以利用它们来简化软件开发工作。但是，许多使用设计模式的人未能正确评估特定设计模式对他们所面对的特定问题的适用性，试图在完成领域分析之前就把所有的事情都归并到某个设计模式，或者用一组特定的设计模式来解决所有的问题。而反模式所针对的，就是在软件开发过程中各种反复出现的问题，其中的相当一部分就是将设计模式应用于不正确的环境而造成的。

模式可以帮助你识别和实现有益的过程、设计和代码，而反模式的作用与模式正好相反，它们让你留意软件开发过程中潜在的各种陷阱与危险，这些东西都可能会导致项目的毁灭。本书的四位作者长期从事与软件开发、项目管理和培训相关的工作，都具有丰富的行业经验。他们在本书中以现实主义的态度介绍了数十种常见反模式及其相应的解决方案。根据本书的指导，项目管理者可以避免很多常见的问题，提高开发过程的生产率，更及时地提供更能满足需求的系统。

本书主要由宋锐、肖国尊等翻译。如果广大读者需要对本书的内容进行讨论，可以发送电子邮件至coldmoon75@163.com。此外，参与本书翻译的还有马蓉、焦贤龙、邝祝芳、杨明军、张杰良、肖枫涛、刘齐军、闫志强、韩智文。Be Flying工作室负责人肖国尊负责本书翻译质量和进度的控制与管理。敬请广大读者提供反馈意见，读者可以将意见e-mail至be-flying@sohu.com，我们会仔细查阅读者发来的每一封邮件，以求进一步提高今后译著的质量。

译 者

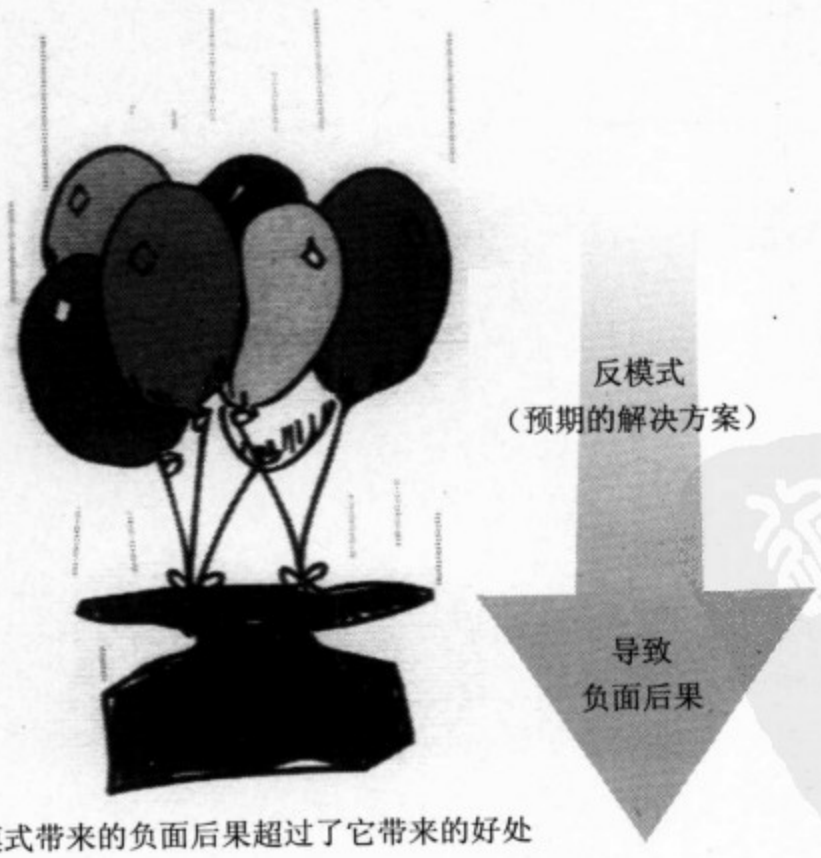
2007年1月于湖南长沙

# 序

我们非常荣幸地受邀来为这本有关反模式的书作序。第一次听别人提到这个术语时，我们更多的是迷惑。除非你已经了解反模式是什么，否则很可能也会感到不解。但在更深入的研究之后，我们发现这确实是一个有趣的主题，并且具有巨大的实际价值。

我们中的大多数人都相当熟悉（或者至少听说过）设计模式的概念，无论是在软件开发环境中还是在别的环境中。设计模式这一术语本身是无须解释的：设计是过去（经过实践）被证实可以取得成功，因此可以成功地复用的。

但什么是反模式？它是某种不是设计模式的东西吗？或者是以前没有被采用过，缺乏稳健设计的做法？还是那些不起作用的东西？使用设计模式时可以通过复用经过试验和测试的设计来节省金钱、时间和精力，而漫不经心地应用反模式则会导致完全相反的后果。




有时模式带来的负面后果超过了它带来的好处

在英文中，pattern（模式）一词可以是：对重复使用的配件的特定排列；用来指导布料裁减的设计或形状；一个模型或样品。但是pattern也可以是一种行为。本书的作者们见证了生活的各

个方面（尤其是软件开发）中行为的负面模式，可能更引人注意的是，这些负面模式在各种规模和层次的经历中都会出现。

反模式可以告诉你要避免哪些情况，而确定要避免哪些情况对成功的软件开发是非常关键的。本书详细说明了与软件相关的反模式的多种类别，并以幽默轻松的方式指出了应该尽量避免的各种软件设计反模式、架构反模式和管理行为反模式。

Jack Hassall / John Eaton

对象管理组织（OMG）

金融领域特别工作组联合主席

# 前　言

阅读反模式或者和朋友一起讨论它们是很有意思的事情。但是不要因此而掉以轻心！本书讲述的是软件技术和开发真实的一面。在本书中，我们定义了在技术和软件项目中实实在在发生的事以及你可以采取的措施。反模式指出了那些会导致劣质软件和项目失败的不良设计概念、技术途径和开发实践。本书还解释了项目中应如何发现和避免这些问题，改善设计与实践，使软件获得成功。

你能从容面对真相吗？真相令人吃惊地难以表达，而且往往被人看作是不正确、不通世故的。真相并不会让所有人都感到高兴。为了使对我们这个行业的揭示更易于接受，我们尽可能地采用了一些喜剧的方法，但是常言道乐极生悲。事实是，软件工程的现状就是一场悲剧：5/6的公司开发项目被认为是不成功的[Johnson 1995]。大部分软件系统交付时提供的功能远少于期望，几乎所有的系统都是烟囱系统，无法适应变化的业务需要。

在试图解释软件上普遍缺乏成功的过程中，我们得出的结论是实践中的反模式的数目远远多于设计模式。在网络时代，技术变化如此迅速，以至于昨天的设计模式会迅速变成今天的反模式。本书介绍了在实践中、在产品中以及在软件文献中反复出现的反模式。然后，我们在每个反模式中都包含了一个重构方案来指出解决方法，该重构方案来自我们在现实世界中使用过的、或者看到过是有效的解决方案。

## 内容概述

本书的第一部分（第1章～第4章）介绍了设计模式和反模式。接着提供了一个反模式参考模型来建立将在反模式说明中使用的共同定义。有经验的读者应该从第2章中的参考模型说明开始。第二部分（第5章～第7章）包含对反模式的说明，是从对开发性反模式的讨论开始的。然后介绍了架构性反模式和管理性反模式。最后，本书的第三部分为更深入地研究和应用反模式提供了资源。

## 反模式与设计模式的关系

反模式是下一代的设计模式研究。它们针对已有实践、遗留设计和前向工程方案，覆盖了更为广泛的应用。

## 补充材料

本书的主页位于www.serve.com/hibc/AntiPatterns/index.htm。作者们会在该网站提供更新。

# 致　谢

虽然我们希望能够感谢所有让本书成为可能的人，但实在无法列出所有为我们提供观点、帮助和鼓励的人，在此特别感谢：

Rohit Agarwal
Don Awalt
Tom Beadle
Pier-Yves Bertholet
Irv Boeskool
John Brant
Frank Buschmann
Bruce Caldwell
Ian Chai
Hugh Chau
Vic DeMarines
David Dikel
John Eaton
Karen Eaton
Ken Kinman
John Kogut
Gary Larson
Steve Latchem
Eric Leach
Dave Lehman
Barry Leng博士
David Lines
Pat Mallett博士
Mark Maybury博士
Dave Mayo
Dennis Egan
Marty Faga
Jack Flannagan
Brian Foote
Alejandra Garrido
Tom Gleeson
Julie Gravallese
Steve Gulick
Patrick Harrison博士
Dan Harter
Jack Hassall
Hebden兄弟
Christy Hermansen
Ruth Hilderberger
John Polger
Don Roberts
Darrel Rochette
Mark Rosenthal
Henry Rothkopf
Bill Ruh
David Samuels
Thad Scheer
Robert Silvetz博士
Bruce Simpson
Theresa Smith
Roy Hiler
Steve Hirsch
Michael Hoagland
Bill Hoffman
Jon Hopkins
Barry Horowitz博士
Bill Ide
Tom Jenkins
Ralph Johnson
Pat Jones
Michael Josephs
David Kane
David Kekumano
Ajay Khater
Kurt Tran
Kevin Tyson
Doug Vandermade
James Van Guilder
Robert Wainwright
Kim Warren
美国华盛顿特区软件图书学习俱乐部
John Weiler
Diane Weiss

Kate Mowbray
Lewis Muir
Diane Mularz
Eiji Nabika
Jason Novak
Jeanne O'Kelley
Ed Peters
Richard Soley博士
Ed Stewart
Shel Sutton
Fred Thompson
Bhavani博士
Thuraisingham
Pat Townes
Anthony Whitson
Jerry Wile
Deborah Wittreich
Joe Yoder
Ron Zahavi
Tony Zawilski

# 本书概要

本书将帮助你识别和克服那些在软件开发过程中普遍存在、反复出现并会影响到软件成功开发的障碍。反模式清晰地定义了大部分人在开发软件时经常会犯的错误。实际上，大部分人犯这些错误的一致性甚至可以达到ISO 9001对一致性的要求！反模式还提供了对这些错误的解决方案：如何解决已经出现的问题以及如何避免在将来重复这些灾难。简而言之，反模式描述和解决了现实世界中的问题。下面这些问题是本书内容的一些例子：

（1）最常见的两个软件设计错误是什么？如何才能识别它们？参阅第5章中的The Blob反模式和Poltergeists反模式。

（2）如何修复（或重构）不良的软件？参阅第5章中的Spaghetti Code反模式和第6章中的Stovepipe System反模式。

（3）设计项目正在原地打转，如何才能让它回到正轨？参阅第7章中的Analysis Paralysis反模式和第6章中的Design by Committee反模式。

（4）我如何才能知道被软件供应商误导了？参阅第6章中的Vendor Lock-in反模式和第7章中的Smoke and Mirrors反模式。

（5）最新的标准或技术突破可以解决问题吗？参阅第6章中的Wolf Ticket反模式和第5章中的Continuous Obsolescence反模式。

（6）软件项目是否正走向灾难？参阅第7章中的Death by Planning反模式和第5章中的Mushroom Management反模式。

（7）软件复用中的常见陷阱是什么？参阅第5章中的Cut-and-Paste Programming反模式和Golden Hammer反模式。

反模式阐明了那些导致开发障碍的负面模式，并包含了经过验证可以把软件开发问题转变成机会的解决方案。反模式可以服务于两个重要目的：帮助识别问题和帮助实现对问题的解决方案。理解问题是进行恢复的第一步。如果要没有确定要解决的问题，解决方案就没有什么用处。反模式有多种原因，并具有相关的症状和后果。我们对每个反模式的这些方面都进行了说明，以澄清为什么要进行改变。然后我们为每个反模式提供了经过验证的、被频繁使用的解决方案。

反模式和另一个重要的软件概念——设计模式密切相关，后者记录了被频繁使用的解决方案。在导致的问题比它解决的问题更多时，设计模式就会成为一个反模式。


图E-1　这是个恶梦

所有的模式都会产生一定的后果。有些情况下，一个模式是可以解决某个问题的好方案，而在另一些情况下，它会成为一个反模式。要想做出可能带有副作用的合理决策，理解模式的这些依赖于上下文环境的后果很重要。我们将从下面这些视角对每个模式进行检查，并说明了反模式在什么时候会提供益处：

- 管理性的（管理过程和人员）。
- 架构性的（定义技术策略）。
- 开发性的（编码实践）。

管理性的、架构性的和开发性的反模式在第5章～第7章中依次进行了定义。如果你刚接触设计模式或者反模式，参阅第1章～第3章提供的介绍性材料。对设计模式实践者，第2章说明了反模式参考模型；第3章说明了使用的模板。这些介绍性的章节有助于让反模式发挥最大的作用。

## 阅读本书的原因

反模式对软件的成功是至关重要的，主要有以下关键原因：

- 反模式随处可见。失败的软件项目远多于成功的软件项目。不良的软件设计、决策和项目远比良好的更为普遍。现实世界中的软件充满了反模式，当然也存在极为高效的解决方案。随着深入学习本书，你将会更清楚地看到这一点。
- 反模式澄清了最常见的软件设计错误。不良的设计和软件是一贯的原因、常见的误解和经典的错误的结果。反模式解释了为什么会出现不良的软件、如何重构不良设计和不良

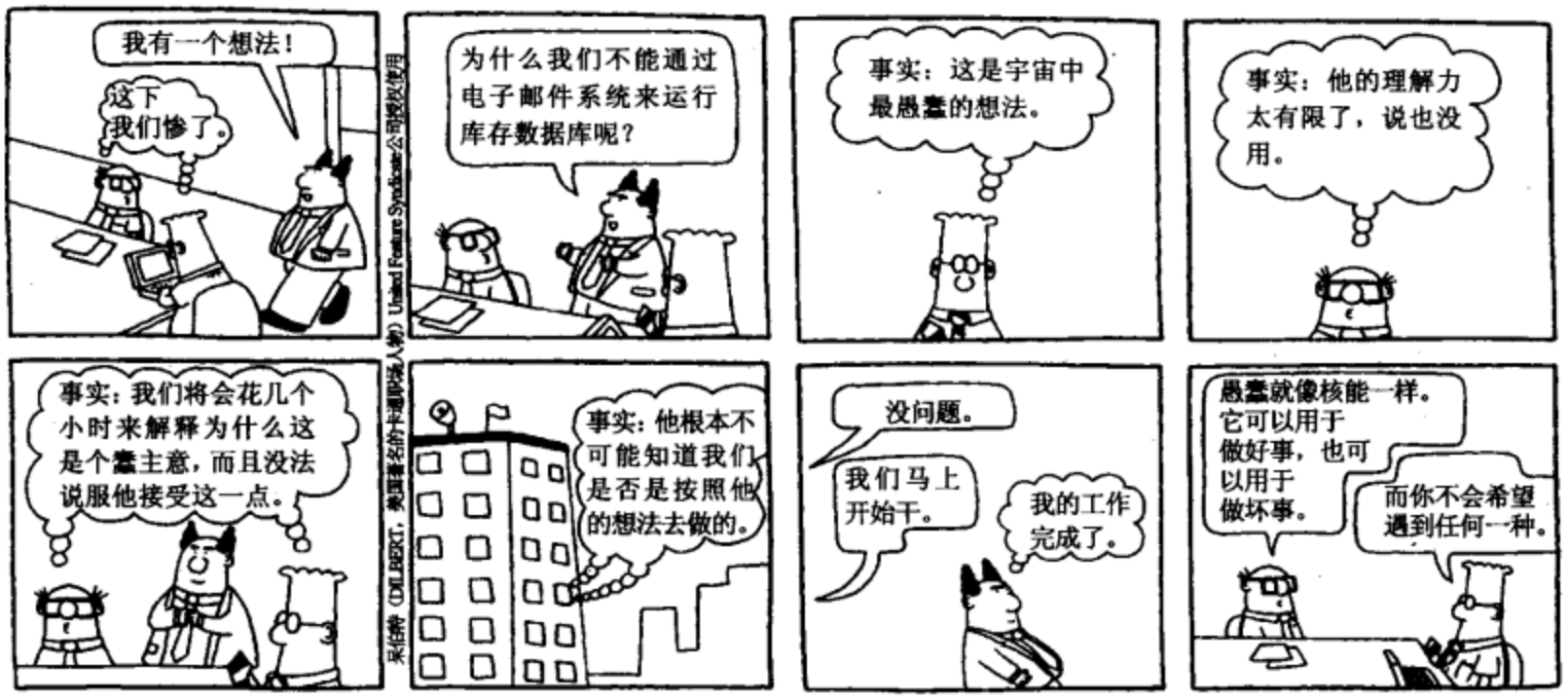
软件，以及如何避免重复这些错误。通过反模式，你可以学会如何及时发现和修复错误以避免严重的后果。

- 反模式揭示了有关软件行业的真相。商业软件技术中充斥着缺陷、矛盾、虚假承诺和反模式。本书从业内人士的角度揭露了有关软件技术的真相。反模式记录了有关商业技术的关键信息，而这些信息对于你适应新兴的商用现货（COTS）驱动的开发范型是必需的。
- 反模式解释了软件项目的现实。大部分软件项目是混乱、不可预计的，而且对职业生涯是危险的。反模式解释了软件项目实际上是如何运作的，以及如何管理它们来避免不良后果。
- 反模式对变化管理是必需的。反模式给负面的软件实践建立了清晰的定义。它们是有用的说明，指出了你所在的机构会希望改变的实践。本书提供了对反模式的全面分类，这可以推动变化管理。
- 反模式定义了重要的术语。每个反模式都为常见的软件实践定义了广为使用的术语。有很多反模式术语用于说明为什么事情会出错。与设计模式相似，反模式通过使用关键短语来指代复杂的概念，所以没有必要重新发明这些概念。使用这些术语的机构和个人可以提高工作效率。
- 反模式是更有效的设计模式形式。我们是设计模式运动的信徒。*CORBA Design Patterns* 一书解决了以模式作为一种文字形式时的一些缺点[Malveau 1997]。本书更进了一步：对模式范例进行了重新定义和扩展，得到了一个全新的形式，也就是反模式。

反模式形式零星地出现在因特网上，它们主要是设计模式社区的作者们编写的非正式反模式。根据网上论坛上的观点，模式社区一致赞同反模式是值得研究的领域。本书对反模式概念进行了全面的发展，所得出的结论是十分有效的问题解决形式。

有一些关键原因让我们相信反模式是有效的。普通的设计模式开始于对上下文环境和作用力的长篇讨论，在逻辑上自然导向特定的解决方案。虽然与为了引出设计模式解决方案所需的长篇文字讨论相比，方案本身似乎更显而易见，但是许多读者发现这种风格冗长乏味。与之相反，反模式一开始是会导致严重后果的。通过强调可能产生的灾难，反模式描述了更吸引人的现实情况而不是抽象的作用力。在此之后提出了一个建设性的解决方案。通过详细说明症状和后果，反模式为要采纳的变化提供了具有说服力的论据，而普通设计模式在这一点上是无法与之相提并论的。而且，反模式可以吸引你的兴趣和想象，而任何设计模式都没有这种魔力。

请带着开放的心态来阅读本书，你会发现学习反模式并与同事讨论它们是很有趣的事。反模式是根据实际软件开发中的成败得到的。我们希望你在阅读本书时能够得到与我们编写和分享反模式时获得的同样乐趣。




图E-2　行政决策造成反模式

# 目　录

# *Part 1* 第一部分

# 反模式绪论

本部分内容

# 第1章 模式与反模式简介

反模式代表了计算机科学和软件工程思想中一系列革命性变化中出现的一个新概念。尽管软件领域已经走过了近50年的开发可编程数字系统的历程，但是在人们把业务概念转换成软件应用程序时，仍然有一些基础问题有待解决。许多最主要的问题来自于开发过程，这些过程需要参与者有共同的愿景和协作来实现系统。大多数已发布的有关软件科学的著作都着眼于积极的、具有建设性的解决方案。本书不同的是，我们的研究将从产生负面影响的解决方案入手。

理论研究者与实践者们设计了数千种构建软件的新方法，从令人激动的新技术到各种渐进性过程。虽然有了这些很好的理念，但实践中管理人员和开发人员取得成功的可能性仍然不容乐观。一项对数百个集体开发的软件项目的调查表明，每6个软件项目中就有5个被认为是不成功的[Johnson 1995]，而且有大约1/3的软件项目被中途取消。未被取消的项目交付时所消耗的成本和时间几乎都达到了最初计划的两倍。①



“系紧你的安全带，今晚的旅程将会充满颠簸。”

——Joseph L. Mankiewicz（美国著名导演）

几乎所有交付的系统都是烟囱系统，也就是无法适应变化的系统。软件最重要的一点就是它的适应性。一半以上的软件成本都是由于需求的变化或者对系统进行扩展的要求所造成的。大约30%的开发成本是由系统构建过程中的需求变化造成的。

## 1.1 反模式就是揭露假象

软件本应该让数字硬件具有更多的灵活性，但是软件技术一直在传播着一系列未能实现的许诺。可以让硬件具有更多灵活性只是这些许诺中的一个。是什么出了问题？在我们的职业生涯中

① 根据Standish集团2007年发布的CHAOS报告。软件项目取消率已从1994年的31.1%下降到2006年的19%；而项目不成功（指时间和成本超过预期、客户需求未完全满足）率从1994年的52.7%下降到46%，虽有进展，但并无太大改观。——编者注

有无数软件开发的时髦技术来了又去。它们在软件开发的特定领域能够取得一定的成功，但是都无法提供本来所许诺能够解决一切问题的“银弹”（见图1-1）。还记得这些主要的开发潮流吗？

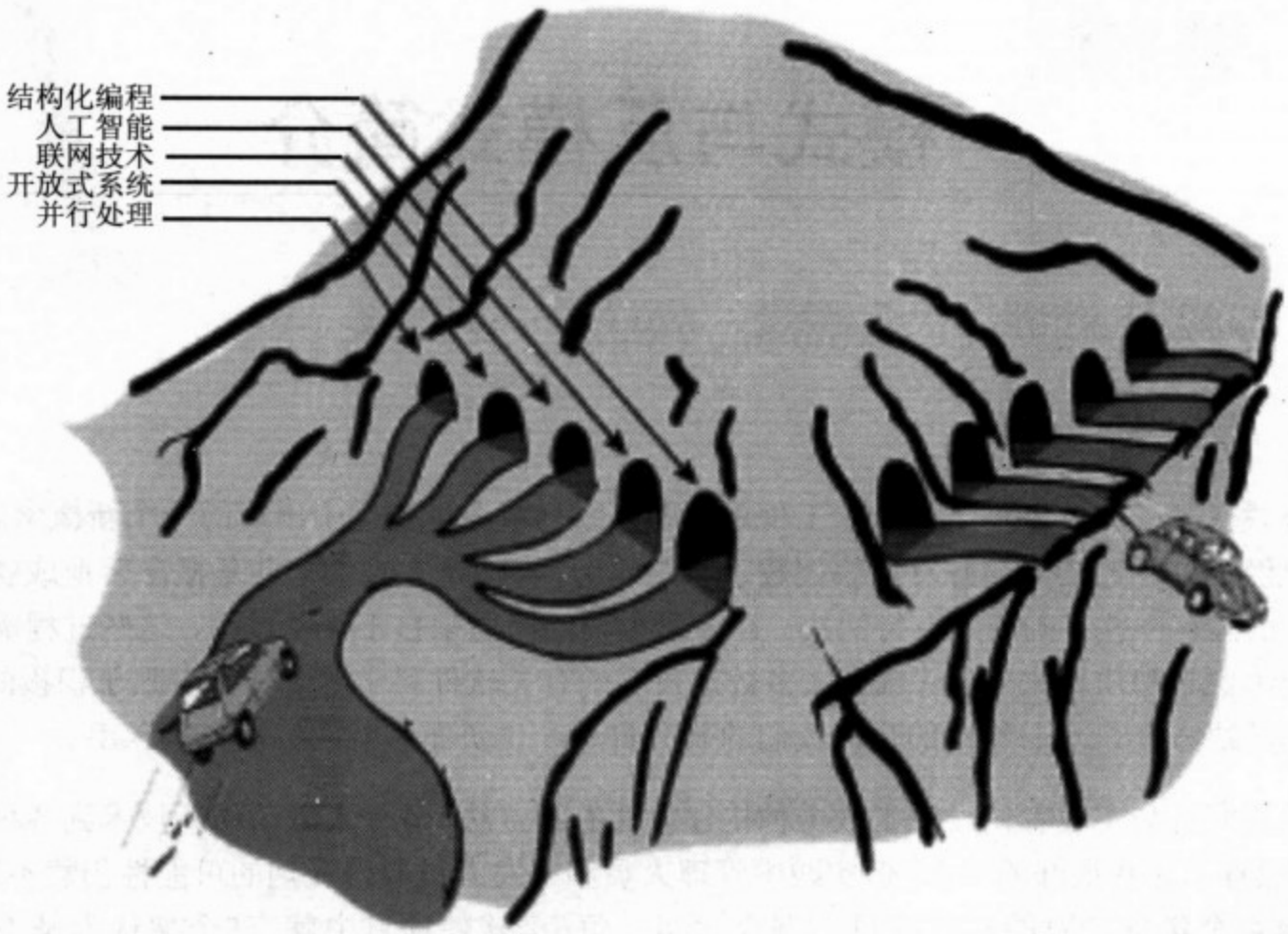



图1-1　各种途径都会走向灾难

- 结构化编程应该提高软件生产率和去除大部分软件缺陷。
- 人工智能应该让计算机具有更强的智力。
- 联网技术应该让所有的系统和软件可互用。
- 开放式系统与标准应该让应用软件可移植、可互用。
- 并行处理应该让计算机具有更强的计算能力而且可伸缩。
- 面向对象应该解决软件生产率和适应性的问题，让软件高度可复用。
- 框架应该让软件高度可复用，让软件开发具有更高的生产率。



这些听起来都像是在放一张破唱片；每次新的软件开发潮流都会做出类似的许诺。现在对许多正时髦的软件技术下定论还为时尚早，但是它们所声称的能力听起来和我们以前曾经听到的非常相似。目前的一些例子包括：

- 因特网。
- 组件软件。
- 分布式对象。

- 业务对象。
- 软件复用。
- 脚本语言。
- 软件代理。

我们要说出有关软件的真相，这个目的非常明确。欺骗与觉醒之间的不断循环似乎已经成为了软件技术的特色，而本书的目的之一就是结束这种循环。



尽早要说明的是，实际上并不只是软件技术供应商们有错。开发人员和管理人员可以采取一些措施来减少软件开发中的问题。无论供应商的宣传是什么，能否取得成功最终还是取决于使用这些技术的方式。

## 软件技术的真相

84%的软件项目是不成功的，而且几乎所有项目交付的都是烟囱系统。为什么会这样？

**供应商会告诉你：**

- 我们的新技术会改变整个软件开发范型。
- 业务对象可以让你的普通开发人员具有更高的生产率（可以用任何术语来替换“业务对象”）。
- 我们会在6个月内提供你需要的所有功能。
- 我们不能以明示或默示的方式提供对我们的软件的适销性或它适用于特定用途的保证。供应商几乎根本不会保证他们的软件能够做任何有用的事。如果他们的软件做了什么不好的事，那也不是他们的错。专有技术只要4~18个月就会改变。快速的技术变化会占据软件维护费用的主要部分，并对软件的开发产生冲击。在客户购买了一个软件许可证之后，供应商期待利用培训、咨询、支持和维护从同一个客户处获得4倍的收入。开发人员遇到的困难越多，供应商能赚到的钱就越多，有没有这种可能呢？

**软件权威会告诉你：**

- 他们的新方法改进了过去的所有问题。
- 他们的工具对包括代码生成在内的软件开发提供全面支持。
- 你需要更多的工具！
- 你需要更多的培训！
- 你需要更多的咨询！

我们这些权威彼此之间并不一致，而且我们的想法也经常会改变。权威们还会推销一些与他们过去向别人推销的想法相矛盾的新想法。权威们的方法学对任何一个实际项目来说都太一般化了。这些方法中的重要细节都隐藏在他们的昂贵产品背后。例如，权威们所宣传的从建模工具生

成具有产品质量的代码的能力，可能还需要好几年的时间才能实现。更常见的是，他们从来就不会说出重要的细节。

重点是，无论有多少这样的激动人心与欺骗宣传，软件技术实际上还处于石器时代（见图1-2）。开发人员、管理人员和最终用户正在为此而付出代价。


图1-2　软件技术还处于石器时代

## 1.2　反模式的概念

反模式是一种文字记录形式，描述了对某个问题必然产生消极后果的常见解决方案。由于管理人员或者开发人员不知道更好的解决方法，缺乏解决特定类型问题所需的知识或经验，或者是在不适当的环境中应用了一个完美的好模式，这些都会造成反模式。在采用适当文字记录下来的时候，反模式描述了一个一般的形式、其主要原因、典型症状、后果，以及一个如何把该反模式改变成更健康状态的重构方案。



反模式是把一般情况映射到一类特定解决方案的有效方法。反模式的一般形式为它所针对的那类问题提供了一个易于辨识的模板。此外，它还清楚地说明了与该问题相关联的症状以及导致这一问题的典型内在原因。这些模板元素完整地说明了特定反模式存在的情况。这个一般形式可以减少使用设计模式时最常见的问题：把特定设计模式应用于不正确的环境。

反模式为识别软件行业反复出现的问题提供了实际经验，并为大多数常见的困境提供了详细的补救措施。反模式突出了软件行业所面临的最常见问题，并提供了一些工具使你能够识别这些问题并确定其内在原因。此外，反模式为逆转这些内在原因的影响、实现有生产率的解决方案提供了一个详尽的计划。反模式有效地说明了可以在不同层次上采取的措施，以便改善应用的开发

过程、软件系统的设计和对软件项目的有效管理。

反模式为辨识问题和讨论解决方案提供了共同的词汇。反模式和与它们相对应的设计模式一样，为开发机构中常见的、有缺陷的过程和实现定义了行业用词汇表。一个更高层次的词汇表可以简化软件实践者之间的交流，并为更高层次的概念提供简练的描述。

反模式在可能的情况下在不同层次上利用机构的资源实现对冲突的全面解决。反模式清晰地声明要把处于不同管理和开发层次上的力量结合起来。软件开发中很多问题的根源其实在于管理层或机构层。因此，不考虑来自这些层次的力量，单纯试图讨论开发性模式和架构性模式是不够完整的。由于这个原因，我们在本书中以大量的篇幅把多个层次上的所有相关作用力集中到一起，来说明核心的问题区域。

反模式以分享软件行业最常见陷阱所带来的惨痛教训的方式提供了一种减轻压力的方法。在软件开发中，识别出有缺陷的情况往往比实现一个解决方案更为容易。在缺乏实现一个反模式解决方案的力量时，屈从于反模式力量所带来后果的个人只能通过了解到在整个行业中还有很多人也和他一样面临两难境地而感到安慰。在意识到某些反模式会导致严重后果时，反模式也可以成为唤醒受害人的警钟，提醒他开始寻找行业中的其他就业机会，写好简历准备另谋高就。

8

## 1.3　反模式的由来

设计模式语言迅速风靡了整个编程界，反映了软件专业人员对改善行业质量与标准的强烈愿望。由于使用和创建可复用设计模式而获得成功的项目在不断增长，设计模式所具有的内在价值是无可争辩的。但是，目前的范型无法完全表达设计模式预期的使用范围，从而导致出现了一种新的文字记录形式，它与现有的模式定义截然相反，这就是反模式。要完整把握反模式在软件开发中的重要性，就要先了解它的来源以及当前的设计模式现象如何导致了它的产生。

**波特兰模式仓库（Portlan Pattern Repository）**

波特兰模式仓库（http://c2.com/ppr/）公布了一个不断进化的设计模式和模式语言的集合。它目前是由Ward Cunningham和Karen Cunningham（Cunningham and Cunningham公司）主办，是一个有趣而令人激动的地方，可以在此通过因特网社区与其他的设计模式迷们进行有关设计模式的合作。在这个网站可以找到反模式的相关资料[①]。

设计模式的思想起源于Christopher Alexander。他是一个建筑师，创建了一种用于城市规划和城市建筑物构造的模式语言。他的模式语言清楚地说明了他对如何进行建筑设计的看法，并解释了为什么某些城镇和建筑比别的城镇和建筑提供了更好的环境。他捕获专家意见的方法很有创新性，因为它让很多原来只有通过多年的城市规划和建筑经验才能得到的“软”属性变得清楚明确。

9

① 网址是：http://c2.com/cgi/wiki?Antipattern。——编者注

1987年，几个处于技术前沿的软件开发人员发现了Alexander的成果，开始使用模式来记录软件开发过程中的设计决策。特别是Ward Cunningham和Kent Beck开发了一种设计模式语言，用于在Smalltalk编程语言中开发用户界面。在接下来的几年中，他们吸引了一些具有类似思想并同样希望使用设计模式来帮助复用软件设计的人，开始在早期的设计模式运动中培养大家的初步兴趣。

把Christopher Alexander的成果应用到软件开发中有什么好处呢？回答这个问题的时候必须考虑到当时软件开发危机的背景。即使在20世纪80年代，具有面向对象软件开发才能的架构师明显太少，无法满足整个行业的需要。而且，理论研究者未能提供解决问题所需的详细知识，也未能设计出能够应对需求变化的动态软件解决方案，这在软件行业中并不稀奇。这些知识需要多年的行业经验才能获得，而行业中的紧迫需要限制了很多架构师，使他们不能花时间来指导缺乏经验的同事。而且，行业中快速的人员流动使得大多数老板不愿意让高级员工把时间花在指导其他人上面，而是让他们去解决自己的关键业务软件问题。这种做法产生了一种迫切而强劲的需求，需要一种可复用的形式来捕获有经验的开发人员的专家知识，让这些专家知识可以被重复用于培训那些缺乏经验的人。此外，还可以在行业层次上应用设计模式，开展有关建立行业范围的设计解决方案的对话，从而允许领域框架在行业层次甚至是全球层次上进行互用。

Christopher Alexander认为，虽然设计物理建筑物所涉及的大部分过程是不同的，但总会有一个通用不变过程隐含于所有其他过程之下，它精确地定义了设计和建造这个建筑物的根本原则。这样的不变过程就是软件开发中的圣杯，因为它们提供了通用的知识框架，从而可以在当前定制的、不能互用的烟囱解决方案沼泽上构建专家软件解决方案，而不是把专家解决方案又变成烟囱解决方案。

设计模式直到1994年才进入面向对象软件开发社区的主流视线。在1994年中期，Hillside Group历史性地主办了第一次软件设计模式方面的行业会议：Pattern Languages of Program Design（PLoP），会上重点展示了一些用于开发软件应用的设计模式和模式语言。特别值得注意的是Jim Coplien题为*A Development Process Generative Pattern Language*的论文，它是首个把设计模式应用于组织结构分析的例子[Coplien 1994]。Coplien的论文立刻帮助模式运动进入了一个新阶段，不仅包括了软件设计模式，还包括了分析模式、组织模式、教育模式以及其他方面的问题。

10

接下来就是现在已经成为软件设计模式经典教材的*Design Patterns: Elements of Reusable Object-Oriented Software*[Gamma 1994]（《设计模式：可复用面向对象软件的基础》，中文版，机械工业出版社）一书的出版。它介绍了一些常见的、实用的软件设计构造，可以很容易被应用于大多数软件项目中。许多面向对象软件架构师都把这本书奉为圭臬。更值得注意的是，很多人发现它所包含的模式似曾相识，自己在先前的软件项目中应用过的构造正是这些模式所描述的。

行业中对它的最初反应是普遍的肯定，因为它把软件开发的词汇和设计的关注从数据结构和编程惯用法的层次提升到了架构层次。而facade（外观）、adapter（适配器）和visitor（访问者）等词语成为了设计讨论中广为人知的术语。软件开发人员常常组织起草根性质的团体，在他们的软件开发项目中应用某些设计模式，并支持其他人对这些模式的使用。世界各地都组织了设计模

式研究团体，讨论把软件设计模式的应用作为提高软件质量的基础。还出现了许多顾问，他们帮助各个机构挖掘机构内部存在的设计模式，以帮助缺少经验的开发人员采用更有经验的同事的技巧。设计模式带来了一个短暂的辉煌时期，似乎成为了推动整个软件行业革命化的一步，让软件行业关注设计复用，设计出能更有效应对变化的需求的软件。

**如何使一个软件项目走向失败**

- 两次向同样的受众展示同一个产品演示。
- 关注于技术而不是问题和场景。
- 未能让投资回报最大化；例如，开发概念验证原型比在已有原型上增加内容更有效。
- 把项目的关注点从大尺度转移到较小的尺度。
- 在多次发布之间不保持一致性。.
- 把开发团队的工作和机构中的其他群组隔离开。
- 重写已有的客户端、服务器端和应用程序。
- 改变系统的目的，以致模型描述的是错误的关注点和对象。

11

从1994年以来，有关设计模式的文献出版量呈指数形式增长。这种增长既有利亦有弊。对有经验的面向对象架构师来说，现在有许多的还在不断增长的可复用设计，可以对它们进行评估并应用到软件开发中去。而且，有大量的论文和研讨会可以帮助架构师把他自己的领域知识记录成设计模式，让行业中的其他专业人员可以更容易使用它们。弊端就是，许多使用设计模式的人未能正确评估特定设计模式或设计语言针对他们的特定上下文环境的适用性。此外，部分开发人员对一些知识一知半解，就急切地想在完成领域分析之前就把所有的事情都归并到某个设计模式，或者用一组特定的设计模式来解决所有的问题。

Michael Akroyd在此期间为1996年的Object World西部会议准备了一篇题为*AntiPatterns: Vaccinations against Object Misuse*的报告[Akroyd 1996]。他的报告基于对行业中面向对象文献的详细分析，试图定义一个有关面向对象的综合观点。报告集中于识别反复出现于多个软件项目中的有害软件构造。这是GoF（Gang of Four，指*Design Patterns: Elements of Reusable Object-Oriented Software*一书的四位作者）模式的直接对照，GoF模式强调的是在构建新软件时使用经过验证的良好设计。

在Akroyd之前，其他一些人也曾提到过反模式的概念，但他是第一个建立起反模式的正式模型的人。对反模式的作用的讨论几乎是和模式的引入同步开始的。Fred Brooks[Brooks 1979]、Bruce Webster[Webster 1995]、James Coplien[Coplien 1995]和Andrew Koenig都曾记录了类似的工作，在确定机能不良行为和重构解决方案的基础上为软件开发提供指引。

由于有如此多的人为反模式做出了贡献，因此把反模式的原创性归功于任何个人都是有失公允的。还不如说反模式是实现设计模式和扩展设计模式模型的工作中会自然发生的一步。本书试图在GoF设计模式学院式的形式主义和那些缺乏经验的软件开发人员之间架起一座桥梁，因为他

12 们需要更多的相关背景信息来评估和判定特定的方法是否适合他们的特定情况。

### 对反模式的研究

> “对反模式的研究是一项重要的研究活动。仅仅在一个成功的系统中展现出‘好’的模式是不够的；你还必须说明这些模式不会出现在不成功的系统中。与之相似，揭示出特定的模式(反模式)出现在不成功的系统中，而不会出现在成功的系统中同样有益。”
>
> ——Jim Coplien（模式先驱，C++大师）

反模式的概念是首个关注于负面解决方案的软件研究方向。考虑到软件缺陷和项目失败出现的频率，负面解决方案也许是具有更丰富研究内容的领域（也就是所谓的目标丰富的环境）。在反模式研究中，我们试图分类、标记和说明反复出现的负面解决方案。我们并不止步于此。对每一个反模式，我们都会给它附加一个或多个设计模式，它们为解决导致问题出现的根源提供了建设性的替代方法。

我们从三个角度来介绍这些反模式：开发人员、架构师和管理人员。

- **开发性反模式**包括程序员遇到的技术问题和解决方案。
- **架构性反模式**确定和解决规划系统结构时常见的问题。
- **管理性反模式**针对软件开发过程和开发机构中的常见问题。

这三个视角是软件开发的基本方面，每一个角度都存在许多问题。

## 1.4　本书组织结构

本章描绘了反模式的发展历程以及它们和改善软件行业状况的关系。反模式提供了一种有效的方法，可以说明在软件开发中发现的存在问题的开发行为和机构组织方法，并详述相应的解决
13 方法。为此，本书如下组织后面各章：

- 第2章介绍反模式参考模型。该参考模型定义了在第二部分对反模式的定义说明中用到的所有常见概念。
- 第3章展示了反模式的两种形式：小型反模式模板和完整反模式模板。
- 第4章解释了反模式在软件开发机构中出现时意味着什么。该章说明了使用反模式时基本的指导原则和行为方式。我们还介绍了自行编写反模式的步骤。
- 第二部分提供了反模式和小型反模式的详细说明。
- 第5章定义软件开发性反模式。
- 第6章定义软件架构性反模式。
- 第7章定义软件项目管理性反模式。
- 第三部分是各种附录，提供了相关的参考资源，包括缩略词表、术语表和反模式摘要。

14

# 第2章 反模式参考模型

本章将概述第5章~第7章中的反模式所使用的参考模型和模板，并介绍与本书中各反模式相关的通用定义和概念。

如图2-1所示，模式和反模式是相关的概念。设计模式的基本要素就是问题及其解决方案。模式中通常会详细阐述问题的上下文环境和影响到该问题解决方案设计的作用力。而解决方案的作用则是以某种方式解决这些作用力，从而导致一些收益、后果和后续的问题。这些新问题又会引致其他模式的适用。模式是在实践中可以经常看到被普遍使用的解决方案。解决方案至少需要使用过三次才能被看作是一个模式。因为没有哪三次使用情况会完全一样，所以设计模式是对这些经验的抽象。

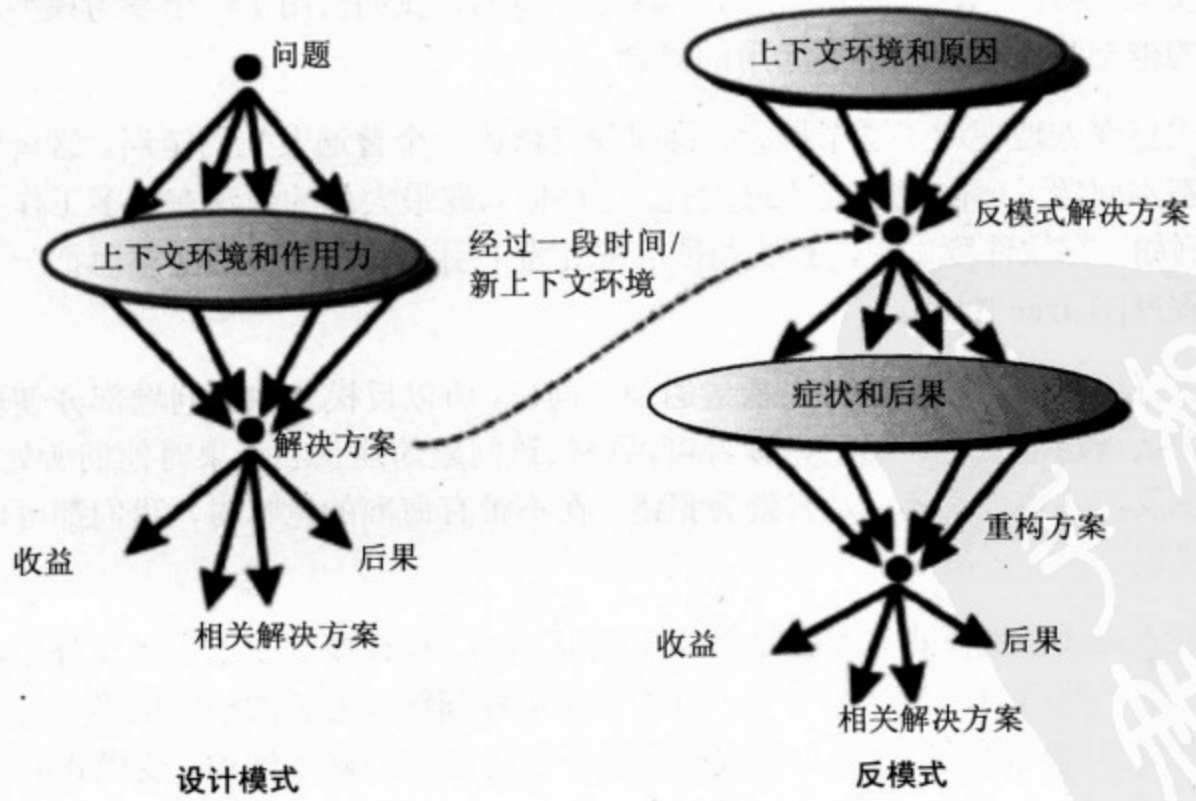



图2-1　设计模式和反模式的概念

设计模式和其他的软件知识表述形式的区别在于对模板的使用。模板就是用于记录模式的一致的大纲，它可以保证模式完整、一致地覆盖了该解决方案、影响设计的作用力和其他后果。模
15 板中包含有用的、预先写好的论据，可以证明该模式的适用性并预测它的后果。

反模式的核心要素是两个解决方案，而不是普通设计模式中的一个问题和一个解决方案。第一个解决方案是*存在问题的方案*。它是某个被普遍使用的解决方案，但是它产生的负面后果占了统治地位。第二个解决方案被称为*重构方案*（refactored solution）。这个重构方案是可以解决该反模式的问题，把它重新处理成更有益形式的常用方法。作为模式的主要支持者之一的Ralph Johnson引导了这一领域的许多工作。要了解更多的信息，访问这个URL：st-www.cs.uiuc.edu/users/droberts/tapos/TAPOS.html。①

模式和反模式是相互关联的。设计模式常常有可能转化成反模式。一个流行的模式，例如过程式编程，可能在某个时期是受欢迎的范例，慢慢地大家更清楚地了解到它所导致的后果而不再受宠。模式解决方案和反模式解决方案之间的区别在于它们的上下文环境：处于不合适的上下文环境中的模式就成了反模式。当一个模式变成反模式时，具备将该解决方案转换成更好方案的能
16 力是大有裨益的。这种变化、移植或者说进化的过程被称为重构。进行重构的时候，我们把一个解决方案转变成改进了结构的另一个方案，而这个结构对系统更为有益。

模式的关键问题之一是它的可读性。对模式的描述通常开始于对上下文背景和作用力的冗长说明。与一开始的长篇讨论相比，后面的解决方案往往看起来是显而易见的。在另一本书*CORBA Design Patterns*中，我们按照让读者可以迅速理解该模式的用途和它的解决方案概念的方法来组织模式的模板，解决了这个问题。为了实现这个目的，我们使用了一个参考模型，用它来定义一个概念上的框架和所有模式中都通用的概念。

反模式更深入地使用了这个概念，将问题描述为一个普遍发生的错误。错误使用的解决方案会把一个基本问题下所隐藏的灾难最大化。这种把问题最大化的做法在很多工作领域都是一种基本做法。例如，在软件测试中，测试人员常常会为了引起开发人员的注意而把一个缺陷放大到系统崩溃的程度[Beizer 1997]。

因为进行恢复的第一步是首先承认遇到了问题，所以反模式中的问题部分使用戏剧化的语言来帮助读者认清这个问题。然后，读者可以评估该问题的症状和后果对他们所处情况的适用性。很多人发现反模式也很有趣。是人就会犯错。在不带有侮辱的意味时，我们都可以嘲笑自己和别人犯下的错误。

编写设计模式和反模式时都是从解决方案开始的。设计模式中的上下文环境、问题和作用力被写成只会惟一地引向一个解决方案。为了保证这种到解决方案的映射的惟一性，设计模式中常常包含对作用力的大段说明。而反模式则是基于与之不同的修辞结构。反模式的说明是从一个引

① 目前有关重构的最佳资源是Martin Fowler的《重构》一书（英文注释版，人民邮电出版社）及其配套网站：www.refactoring.com。——编者注

人注目的、有问题的解决方案开始，然后提供一个替代解决方案来重构该问题。重构方案并不一定是惟一的；它是可以解决问题、提供更多好处的有效方法。模板中有一个独立的部分包含了对解决方案的所有重要变化。

实际上，我们发现在描述重复出现的解决方案时，反模式是比设计模式更有力且更有效的形式。我们希望你也会发现这一点。反模式起始于一个已有问题的解决方案（或者说遗留的方法），该问题对大部分已有模式来说都是绿地问题（需要从头编程）。我们发现在实践中具有遗留问题或已有问题的情况更为常见。反模式放大了这个问题，帮助开发机构来识别出问题的结构、症状和后果。然后，提出一个通用的解决方案来重构系统，以提高收益，尽量减少不利的后果。 17

## 2.1　视角

本书主要从三个视角来介绍反模式：开发人员、架构师和管理人员（如图2-2）。开发性反模式描述的是程序员在解决编程问题时遇到的情况（参阅第5章）。架构性反模式关注的是系统结构方面的常见问题、它们导致的后果及其解决方案（参阅第6章）。软件系统中许多最严重的待解决问题都出现在这个方面。管理性反模式说明的是软件开发机构中出现的常见问题及其解决方案（参阅第7章）。管理性反模式会影响到在软件开发中担任角色的所有人，对它们的解决方案会直接影响到项目在技术上能否成功。

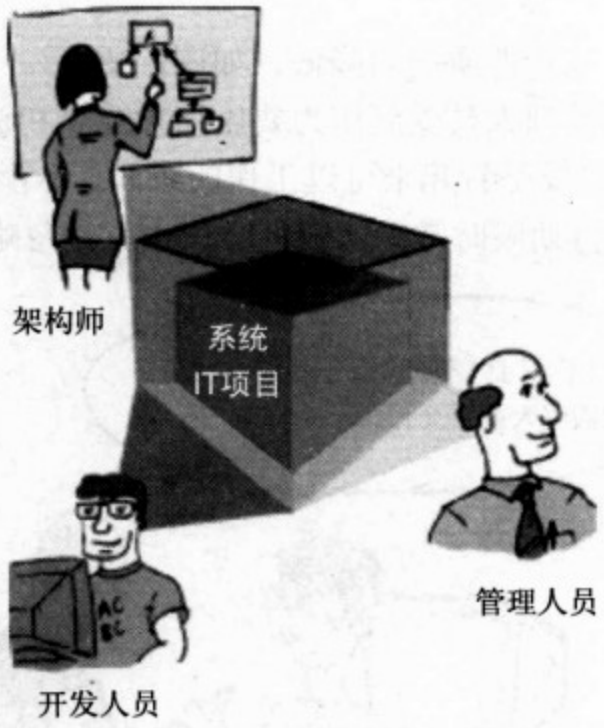



图2-2　主要的反模式视角

我们将利用一个参考模型来说明3个视角中通用的术语。该参考模型的主体是用于介绍反模式关键概念的3个主题： 18

- 根源。
- 原力。

- 软件设计层次模型。

根源（root cause）提供了反模式的基本上下文环境。原力（primal force）是影响到决策制定的关键激励因素。根据架构规模的不同，这些力量的相对重要性会发生很大的变化。架构规模则是通过软件设计层次模型（Software design-level model，SDLM）来定义的。对架构层次的理解定义了模式适用的不同规模。每个模式都有一个最适用的规模，第5章～第7章就是在这个最适用的范围上来介绍这些它们，但是它们也可能在一定程度上适用于其他的规模（在每个模式中也进行了说明）。模式模板提供了用于模式定义的大纲。

## 2.2　根源

根源就是在软件开发中会导致项目失败、成本超支、进度落后和无法满足业务需要等问题的错误[Mowbray 1997]。本节所指出的根源是非常普遍的：所有软件项目中有1/3被取消；每6个软件项目中有5个被认为是不成功的[Johnson 1995]。不幸的是，面向对象技术并没有改变这一总体趋向。实际上，每次新技术潮流（例如客户机/服务器技术）都会趋向于增加软件风险，增加这些根源导致失败的可能性。项目失败的根源被归结为“七宗罪”。这个类比成功地指出了那些低效的实践方法[Bates 1996]，它得到了广泛的接受。

### 2.2.1　匆忙

匆忙（haste）的决策会导致软件质量的降低，如图2-3所示。软件项目在进度方面常常会承受很重的压力。在项目起始时，管理人员受到压力要压缩预算表和进度表来达到不切实际的目标。随着连续错过项目最后期限，只要是看起来可以工作的东西，不管质量如何都会被认为是可以接受的。测试工作是错过软件交付期限时常见的牺牲品。也就是忽略了下面一些测试工作：


图2-3　匆忙导致废品

- 单元测试覆盖所有构件。
- 反复测试集成构件的成功路径和错误状态。
- 回归测试。

19

在这样的环境中，为了方便而牺牲掉了长期的架构性利益。

高质量的面向对象架构是认真仔细的研究、决策制定和实验的产物。面向对象架构过程在最小的程度上也需要包括对需求的探求、对架构的挖掘和获得实际经验。理想情况下，面向对象架构包括一组高质量的设计决策，这些决策对系统的整个生命周期都是有益的。

架构师具有大量的领域经验是非常重要的，因为面向对象架构完全是定义好，并且受到保护不改变的。利用适当的领域经验和设计模式，可以迅速定义出高质量的面向对象架构。不过，匆忙地制定面向对象架构决策往往是错误的做法。

### 2.2.2　漠然

漠然（apathy）是指不关心解决已知的问题。虽然并不是所有解决方案都是已知的和可以实现的，但漠然是从根本上就不愿意尝试一个解决方案（见图2-4）。对面向对象架构的漠然会导致系统中缺乏划分（partition）。面向对象架构的一个关键方面就是适当的划分。例如，面向对象架构会把系统划分成各种类并定义它们的接口和连接。

20


图2-4　漠然是恭维的最差形式

面向对象架构中关键的划分决策出现在稳定的可复用设计和可替换设计之间。稳定的设计在系统的整个生命周期中都会一直存在于系统中，而系统中个别的软件模块则可能会被修改、替换或增加。最好把可替换的设计细节放置到配置文件（profile）、纵向特异性相关文档（vertical specialization）和元数据（metadata）中。

忽视关键的划分工作意味着必须改变架构的核心来应对子系统层次的变化。因此，子系统层

次的变化会影响到整个系统中的所有模块。这样，漠然之罪会导致对变化缺乏支持。此外，缺乏良好划分的架构还会使应用系统的互用和复用都很困难。

### 2.2.3　思想狭隘

思想狭隘（narrow-mindedness）就是拒绝实践众所周知有效的解决方案（见图2-5）。一个例子就是在软件系统中对元数据的使用。元数据是软件系统中自我描述的信息，它可以让系统动态地发生变化。


图2-5　无可救药的人

很多面向对象系统在构建时实际上并没有使用元数据。如果不使用元数据，那么应用软件对绑定、关系和有关系统配置的设想的实现都是硬编码的。例如，通过简单地使用元数据服务就可
21 以让服务器和客户机的数目及位置成为透明可变的。CORBA标准中包含大量的公共元数据服务，例如命名服务（Naming Service）、交易服务（Trader Service）和接口存储库（Interface Repository）等。

### 2.2.4　懒惰

懒惰（sloth）是那些图省事的开发人员或管理人员的“健康标志”，他们根据一些“容易得到的答案”制定出拙劣的决策（见图2-6）。分布式对象技术允许应用开发人员使用ISO接口定义语言（Interface Definition Language，IDL）来迅速地定义出系统层次的接口。自动生成的接口代码占位程序（stub）和骨架（skeleton），让构建分布式系统的工作变得相对轻松。容易建立和改变接口导致了懒惰之罪——缺乏配置控制。

图2-6　懒惰通常终结于突然的清晰

虽然懒惰更常见于小规模的面向对象项目，但频繁改变接口的习惯却是难以克服的。对接口 22
的改变越多，接口的语义对开发人员就越不明确。最终，开发人员和维护人员需要花掉一半的工作时间来试图理解系统是如何工作的。而在达到这一点之前很久的时候，系统就已经失去了有关架构的所有观念。

正确的配置控制是从建立原型这第一个阶段就开始的。理想情况下，系统层次的接口在活跃的软件开发过程中是保持稳定的，很少会被改变。稳定的接口可以允许并行开发、有效地编写文档，并减少软件被废弃的可能。

### 2.2.5　贪婪

贪婪（Avarice）可以有很多种形式，但都会导致不恰当的软件开发决策。架构性的贪婪意味着对过多的细节建模，从而由于抽象工作不足而引致过高的复杂性（见图2-7）。

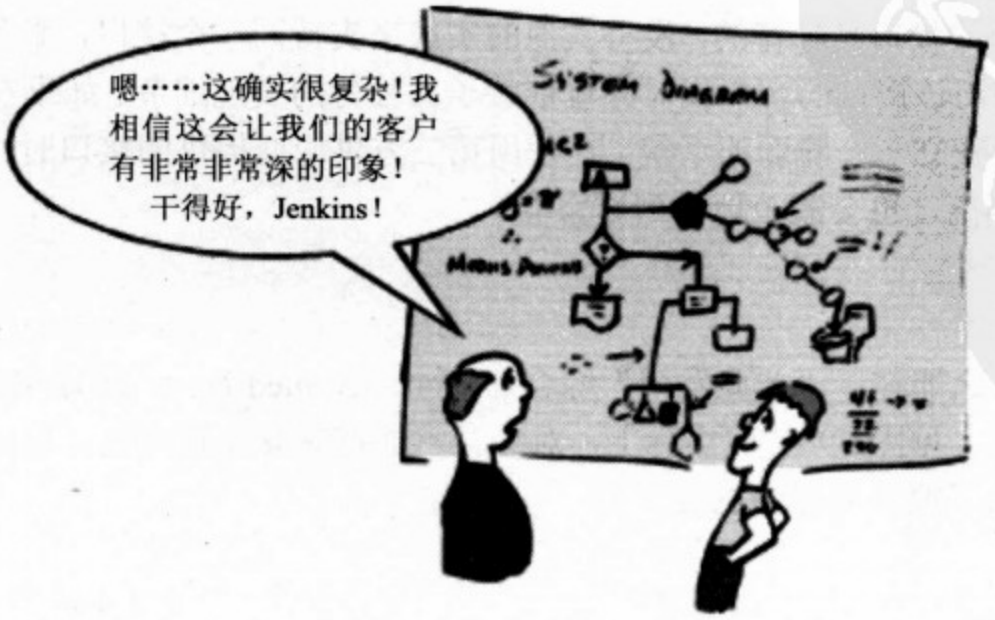



图2-7　对复杂性上瘾

过高的复杂性会导致很多软件问题和对项目的挑战。过度复杂系统的开发、集成、测试、文档编写、维护和扩展的费用是很昂贵的。某些时候，为了弥补在时间和金钱上的损失，会跳过一
23 些开发步骤（例如测试）。这会迅速导致项目的失败。

### 2.2.6　无知

无知（ignorance）是在智力上的懒惰，它是未能寻求到理解的结果。它让人变得愚蠢（见图2-8），并最终引发长期的软件问题。无知之罪（对实现的依赖）常常发生在把应用移植到分布式架构时。由于无知，有些人会设想系统层次的接口是从对已有应用对象的细粒度定义中抽取出来的。例如，在从已有的C++头文件通过反向工程得到IDL接口时，就会建立依赖于实现的接口并在整个系统及其子系统中传播它们。


图2-8　某些人对变化太无知

如果某个对象的接口是独特的，没有其他的实现来支持同一个接口，那么这个接口就是依赖于实现的。所有使用该接口的客户或对象就依赖于其独特的实现细节。如果在系统范围内反复发生这一问题，就会建立一个脆弱的系统。在使用第三方供应商提供的接口时，如果没有通过适当的分层和封装来隔离，也会清楚地看到这宗罪。

### 2.2.7　自负

自负（pride）之罪在于“非此发明”综合症（not-invented-here，简称NIH，即认为不是自己设计就不是最好的，见图2-9）。而实际上，对象技术通过对商业软件包、自由软件和封装后的遗
24 留应用的集成提供了很多复用的机会。


图2-9　自负之后就是惨败



开发人员常常会在本可以通过架构挖掘就能应用已经存在的系统、产品和标准中的知识时，不必要地发明一些新设计。重复发明会引起很多不必要的风险和成本。新软件需要经过设计、编码、调试、测试和文档化。新架构必须先建立原型并逐渐进化才能被证实能够为软件提供好处。

## 2.3　原力

在软件设计中需要进行选择。例如，设计软件架构时出现的一些关键选择包括：

- 哪些细节需要暴露出来，哪些细节又需要抽象。
- 包括和不包括哪些功能。
- 哪些方面需要具有灵活性和可扩展性。
- 哪些方面需要加以限制和保证。

软件设计选择常常是相当复杂的，需要考虑大量的因素（即作用力），例如安全性、成本、适应性、可靠性，等等。弄清决策的上下文环境对做出良好的选择非常重要。可以通过多种方式来澄清设计选择，例如：

- 隔离各种关注点。
- 建立优先级。

要隔离各种关注点，我们需要限制每种选择的范围。可以使用软件架构中的划分来分配和描绘各种关注点的边界。每个划分负责解决有限的一些事项，从而简化决策的制定。架构代表了所有划分的联合，覆盖所有的相关事项。这种对各种关注点的隔离是架构承担的基本角色。

也可以通过对优先级的理解来澄清决策。如果我们知道什么重要什么不重要，那么选择在设计中包含或不包含哪些内容的时候就会容易很多。决策过程很困难，因为它们包含了一些事项，排除了很多其他的事项，而我们必须能够证明这些选择是正确的。这就是架构的另一个基本角色，即解释重要的决策和设计选择。

26 风险是总会出现在软件决策中的作用力。软件项目令人吃惊地容易失败。前面提到过，所有软件项目中有大约1/3被取消，只有大约1/6的软件项目被认为是成功的。未被取消的项目通常会超出预算和计划时间2倍以上。不成功的项目就无法交付需要的功能。系统一旦交付，要改变系统就会涉及很高的风险。修正或扩展系统都可能导致新的软件问题。

考虑到这些统计数据，每6个软件项目中有5个会走向失败。这些数字基本上没有被诸如客户机/服务器和面向对象之类的新技术和方法所改变。作为软件专业人员，前景是严酷的，除非发生一些重大的变化。我们相信，必须对软件系统的架构方法和风险管理方法做出重大的改变。

我们把风险看作一种一般性的力量，是大多数其他力量都蕴涵的因素。风险管理是在不同程度上激发本书中的模式和解决方案的普遍力量。

## 原力的含义

作用力就是在制定决策的上下文环境中存在的关注点或问题。在设计方案中，被成功处理（或者说解决）的作用力会带来收益，而未被解决的作用力则会导致不良后果。对任何软件问题，都有一些会影响到给定解决方案的力量。对设计模式的应用会引向以特定方式来解决这些力量的解决方案。在这个解决方案中，对某些力量的解决会比对其他力量更为彻底。选择一个设计方案就会建立起作用力的优先级，具有最高优先级的力量解决得最彻底。

某些力量是领域特有的。领域特有的力量（也称为*纵向力量*，vertical force）是要处理的领域或问题所带来的，只出现在特定场景中。由于一个软件解决方案中的纵向力量是独特的（或者说局部的），对纵向力量的解决通常会对每个软件问题产生独特的解决方案。

另一类力量是*横向力量*（horizontal force），在多个领域或者问题中起作用。横向力量是在多个软件模块或构件中都会影响到设计选择的那些力量。对于横向力量，局部的设计选择可能会直接或间接地受到其他地方所做出的设计选择的影响。例如，如果横向力量是“设计一致性”，那
27 么就需要协调多个软件模块中的软件设计来保证这种一致性。

在软件架构和开发中有一类横向力量非常普遍。这种力量在几乎所有的设计环境中都会出现，应该被看作是驱动大多数解决方案的上下文环境力量的一部分，它们被称为*原力*（primal force）。原力的作用之一就是保证架构和开发活动的正常进行。例如，某个软件决策可能本来看起来是局部的，但是当同一个企业中的其他软件开发团组做出了相矛盾的选择时，它可能就会累积产生不利的影响。原力代表的就是由于软件决策之间的相关性而产生的，在软件架构和开发活动中普遍存在的力量。

原力是本书所介绍的模式语言中提出的指导原则的重要组成部分。每种原力都横向作用于软件架构和开发活动的许多领域。原力代表了基于常识的基本考虑，它们对软件架构和开发活动的成功是不可或缺的。原力为软件架构师和开发人员提供了一个价值体系，这一体系独立于作用于特定情况中的那些力量。

原力包括：

- 功能管理，满足需求。
- 性能管理，满足对操作速度的要求。
- 复杂性管理，定义需要的抽象。
- 变化管理，控制软件的发展。
- IT资源管理，控制对人员和IT制品的使用和实现。
- 技术转移管理，控制技术变化。

在不同的层次上，各种原力具有不同的相对重要性。在应用层以及更细的粒度上，功能和性能是关键的作用力。而IT资源管理和技术转移管理涉及的则是企业层和全球层。在我们开始完整地讨论有关内容之前，需要先利用一个可伸缩性模型来定义这些不同的软件开发规模。

表2-1给出了各种力量在不同规模层次上的影响程度：

- 关键的（critical）。基础性的影响，对整个软件产生影响。
- 重要的（important）。要认真考虑该影响，因为它对软件的很大部分产生影响。
- 边际的（marginal）。常常可以忽视该影响，因为它只影响软件中的很小部分。
- 不重要的（unimportant）。不用考虑该影响。

28

**表2-1　影响的不同程度**

| | 全球行业 | 企　业 | 系　统 | 应　用 |
|---|---|---|---|---|
| 功能管理 | 不重要的 | 边际的 | 重要的 | 关键的 |
| 性能管理 | 重要的 | 重要的 | 关键的 | 关键的 |
| 复杂性管理 | 重要的 | 关键的 | 重要的 | 边际的 |
| 变化管理 | 不重要的 | 关键的 | 关键的 | 重要的 |
| IT资源管理 | 不重要的 | 关键的 | 重要的 | 边际的 |
| 技术转移管理 | 关键的 | 重要的 | 重要的 | 边际的 |

- 功能管理（management of functionality）最好在应用层加以解决。开发人员更善于在最低的层次针对（功能性）需求实现功能。
- 性能管理（management of performance）最好在应用层和系统层加以解决。系统层的性能通常涉及在领域模型中进行粗粒度的优化。应用模型将使用这个领域模型，进行局部的、细粒度的优化。
- 复杂性管理（management of complexity）在所有规模层次上都是重要的。不过，在较高

的规模层次上，复杂性会以指数形式恶化。任何时候，复杂性在任何层次上都是需要认真考虑的。

- 变化管理（management of change）在企业层和系统层上是关键的，这时单个产品的变化率处于适中的水平。应用程序和外部关注点的变化相当迅速，从而产生新的需求和对已有实现的改变。因此，在系统层和企业层规划可调整的、能够对变化进行管理的系统很重要。同时不可否认的是，在全球层上的变化更为迅速，但是由于没有哪个开发机构能够对此加以解决，所以对全球层变化的考虑要少一些。
- IT资源管理（management of IT resource）在企业层是关键的，因为它涉及战略规划的需要。它包括了对人员、时间、硬件和软件制品的管理。它对于在系统层保证主要软件开发活动的成功同样重要。

29

- 技术转移管理（management of technology transfer）在全球行业层是很重要的，这样才能跟上技术的进步，跨越机构的界限对经验和知识进行复用。重要的是在企业层让可用的资源发挥出最大的价值，从而在系统层上占据软件发展的技术优势。

表2-2指出了软件开发中的不同角色，以及他们在不同层次上承担责任的程度。每个角色都会产生一个关键的影响，因为他在相应的层次上可以发挥最大的作用，如表2-3所示。

**表2-2　不同角色在各层次承担责任的程度**

| | 全球行业 | 企　业 | 系　统 | 应　用 |
|---|---|---|---|---|
| 信息总监 | 关键的 | 关键的 | 边际的 | 不重要的 |
| 项目经理 | 不重要的 | 关键的 | 重要的 | 边际的 |
| 架构师 | 边际的 | 重要的 | 关键的 | 重要的 |
| 开发人员 | 不重要的 | 边际的 | 重要的 | 关键的 |

**表2-3　在各层次上起作用的角色**

| | 全球行业 | 企　业 | 系　统 | 应　用 |
|---|---|---|---|---|
| 功能管理 | | | 架构师 | 开发人员 |
| 性能管理 | | | 架构师 | 开发人员 |
| 复杂性管理 | | 项目经理 | 架构师 | 开发人员 |
| 变化管理 | | 项目经理 | 架构师 | |
| IT资源管理 | 信息总监 | 项目经理 | | 开发人员 |
| 技术转移管理 | 信息总监 | | | |

30

开发人员主要关心的是对功能和性能进行优化。不过，软件项目要取得成功，同样需要在这一层次对变化管理这一原力加以处理。对架构师而言，需要考虑与开发人员相同的问题，同时还需要处理对整个系统的复杂性管理这一作用力。不管系统组成部分的复杂性如何，架构师都必须把系统设计成具有可管理的系统接口。项目经理要辅助架构师进行复杂性管理和变化管理，并成功完成对IT资源如人员、时间和预算的管理。最后，信息总监要制定计划来管理机构内部IT资源，

以及管理向行业中其他开发机构的技术转移或者从其他机构来的技术转移。

**功能管理**

功能管理用于保证软件满足最终用户的需求。软件提供了从最终用户对象世界到计算机技术对象世界的映射。软件的功能为这种映射和在技术对象上进行的所有操作提供了运行机制。

互用性是功能管理的重要组成部分，由软件模块之间信息和服务的交换构成。互用性允许多个软件模块协同工作以提供功能。

**性能管理**

第二种原力在某些时候会被软件架构师所忽视，它就是性能管理。软件只满足功能需求是不够的；系统还必须达到对性能的要求。最终用户在系统整个生命周期中的认识是变化的，这会对性能的要求产生影响。对系统的一个隐含要求就是它至少应该和采用其他技术开发的可比系统一样快。

在CORBA中，一个开包即用的ORB产品的性能只能达到实现它时所采用的内在技术的水平。ORB内在的客户－服务解耦（decouple）机制允许聪明的开发人员进行很多性能改进而不用改变应用软件。由于性能改进对应用软件是透明的，所以可以经过较长的时间逐步采用，或者根据系统成长的需要来增加。它的结果就是CORBA系统具有高度的可伸缩性，这在许多公司从系统原型到企业范围的运作实现的迁移中都得到了证明。CORBA支持的最著名的性能改进是负载均衡。CORBA使用动态绑定来把客户机连接到服务器，从而有可能在绑定过程中插入算法来保证最优化地使用服务。由于可以用多个实现来支持同一个接口，所以往往可以通过用多个实现复制服务来优化负载均衡。



性能管理还包括应用软件优化。应用的具体实现控制了处理的细节，而这里提供了最多的灵活性来对应用的性能进行调节。大多数性能问题都涉及与计算相关的应用瓶颈，而不是输入/输出性能或者网络性能。特别是应用开发人员控制了应用程序数据结构的选择、算法的设计，往往还有语言的实现。性能优化是一项昂贵的、耗时的活动，很少会有项目愿意承受提高应用程序速度所需的额外的、常常相当可观的成本。

**复杂性管理**

开发系统的时候，牢记良好的软件抽象的价值是相当重要的。抽象可以带来更简单的接口、统一的架构和改善的对象模型。缺乏有效的抽象会导致过度的系统复杂性。软件构件之间的共同性常常没有得到认识和充分的利用。缺乏适当的设计抽象，就会形成不必要的构件差异，导致冗余的软件和为基本相似的代码产生多个维护点。

复杂性管理大致就是对设计进行分析，正确地确定最可能受到未来变化影响的热点和问题区域。例如，预测在某个实现中潜在可能产生性能瓶颈的变化。一旦正确识别了热点，就可以采用

递归的重设计修订过程来提供简单性和健壮性。建立抽象来简化各个接口可以在整个系统设计中节省成本，既包括在单个软件构件的内部实现中，也包括在每个访问该构件服务的客户机上。

**变化管理**

可适性（adaptability）是大家高度渴望但又难以控制的一项软件特性。大多数开发机构都希望获得具有可适性的系统，但是很少有人会认识到开发灵活的系统的完整含义。开发分布式系统
32 的时候，仅仅把可适性当作一个目标是不够的，系统架构师还要考虑系统的发展并决定系统的哪些地方应该如何具有可适性。系统架构师设计接口规范时，他是在做出决策，判定哪些地方具有最高的可适性，而哪些区域具有最高的稳定性。可以使用IDL来定义软件的边界。通过合适地说明，这个边界就是实现系统中构件之间解耦合的接口集。为了让多个实现都能够满足接口的限制，良好的IDL接口集要说明软件构件对外提供的功能。系统的稳定性和可适性是通过软件接口来实现的。支持改变构件的具体实现同时还能维持稳定性的系统，比为了支持新构件实现必须经常修改接口的系统具有更高的可适性。最小化客户机对特定构件实现的依赖是软件架构师的责任。通过抽象化各类构件的关键功能元素并以独立于实现的方式来定义它们，可以达到这一目的。IDL是一种理想的标注方式，因为它是独立于开发语言的。对于ORB，IDL还可以提供对位置独立性和平台独立性的支持。

可移植性（portability）是变化管理的一个重要方面，因为它代表的是把应用软件从一个产品或平台移植到另一个产品或平台的能力。有很多标准通过促进可移植性来降低风险。不幸的是，COTS（commercial off-the-shelf，商用现货）产品提供的可移植性是不完善的，因为供应商会增加或减少对其他COTS和平台的支持。

**IT资源管理**

IT资源管理涉及管理企业大量资产的能力。典型的大规模企业有很多不同类型的硬件（硬件多样性）、很多不同的软件产品（软件多样性）和每种技术的多个版本。在一个变化的机构中管理大量的机器设备和软件，这本身就是一个主要的问题。IT资源管理涉及很多方面，例如软硬件的采购、库存、培训、维护、升级和支持。

大部分最终用户不能自己提供技术支持，所以提供这种支持的负担就落在了企业的肩上。如果不解决支持负担，损失的时间和生产率就会产生大量的成本，根据*Information Week*估计，每台最终用户个人计算机每年可以达到40000美元（1996年4月）。

安全性是IT资源管理的另一个重要方面。随着系统联网和互用性的增加，对信息和服务的安
33 全控制也越来越重要。

**技术转移管理**

技术转移管理包含作用于企业外部边界上的一些关键力量。它包括通过对软件和其他技术的使用与转移而建立的正式的和非正式的关系。由于因特网的普及性和实用性，技术转移管理也是

影响到很多软件开发人员的问题。利用电子邮件、万维网和其他服务，跨越企业界限在全球范围内传播技术信息变得相对容易。如此方便的信息交换会影响到对知识产权的控制，并影响到内部系统对外部技术的依赖性的变化。

技术转移管理还包括建立标准和对标准施加影响的可能性。本质上，标准是企业之间的技术协议；它们代表的是为了建立共同性和合作而在机构之间需要转移的最少技术。CORBA IDL实际上让所有的架构师和开发人员都可以参与接口规范的建立过程。同样的IDL成为了正式的标准制定团体和其他合作团体及联盟所接受的标注方式。现在，大多数机构可以通过为软件接口建立技术协议的方法来管理技术转移环境。

## 2.4　软件设计层次模型

如果试图在没有总体架构的情况下以零碎部分为基础来开发一个系统，那么随着系统由于需求变化和采用新技术而发展，它会逐渐难以管理。架构带来的关键好处之一就是对不同关注点的隔离，也就是说，把问题划分成多个可解决的元素而不是同时处理所有问题。在本书的模式语言中，我们提出了可伸缩性模型来根据软件解决方案的规模隔离不同的关注点。该模型阐述了软件系统中内在的关键层次和每个层次上的问题及可用的解决方案。 34

在典型的小软件系统中，有两个规模层次（见图2-10）。第一个层次是外部模型（或者称为应用层），它直接针对最终用户的需求。这一层次包括用户界面以及相关的功能。应用程序一般由交互式用户控制通过图形用户界面（GUI）或者用户命令来驱动。应用程序实现的是直接与人交互的系统的外部模型。应用程序包含了解决机构的功能需求所必需的软件。

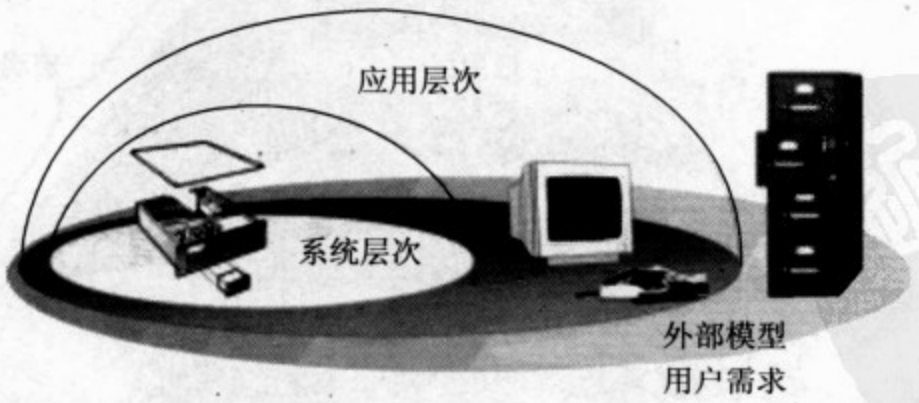



图2-10　软件系统中外部模型和内部模型的同时存在说明需要在不同层次对软件进行考虑

第二个层次是内部模型（或者称为系统层）。系统层由应用程序之间的联系组成，它并不直接接触最终用户，也不会被最终用户直接看到。系统层为软件系统提供了架构。这一层次负责为应用层提供基础结构，并建立应用之间的互用能力、通信和协作。对数据存储的访问、审核和对进程间资源的管理也发生在系统层。

在软件实现的其他几个层次也存在类似的划分。例如在软件解决方案跨越机构中多个系统的时候，在企业层有另一组相关的关注点。可伸缩性模型解释了每个规模上不同的优先级顺序，而

本书中的模式语言包含了一组可能的解决方案。这解决了对指导原则的一个关键挑战：保证将适当的解决方案应用于正确的层次，以最大化开发有效的、可维护系统的可能性。

本书中的模式语言是按照架构层次来组织的，它定义了一个全面的框架来研究面向对象架构中的模式和原则。虽然我们确定了7个架构层次，但重点将处于面向对象架构（见图2-11）中较大规模的层次上。其他一些著作在不同程度上覆盖了较小规模的层次。尤其是对象层是当前的可复用构件库和标准，如C++、Smalltalk、CORBA对象模型和CORBAservices等所解决的问题。在微架构层，GPL（Gamma Pattern Language）[①]和其他的设计模式研究提供了开发构件微架构所必
35 需的结构。在宏构件（框架）层也进行了大量的工作。例如，Taligent公司在开发面向对象框架时，在发展宏构件层的软件和指导原则方面相当活跃。到目前为止，更高的架构层次基本上被忽略了，其后果就是由于只有专有的解决方案和不可复用、不可伸缩的技术解决方案，跨应用、跨系统和跨企业机构的一般互用性受到了严重的影响。通过定义可伸缩性模型，设计模式的领域可以扩展应用于更大规模的问题。在此之前，这些问题已经耗费了太多的资源却没有产生多少可复用的、可扩展的解决方案。

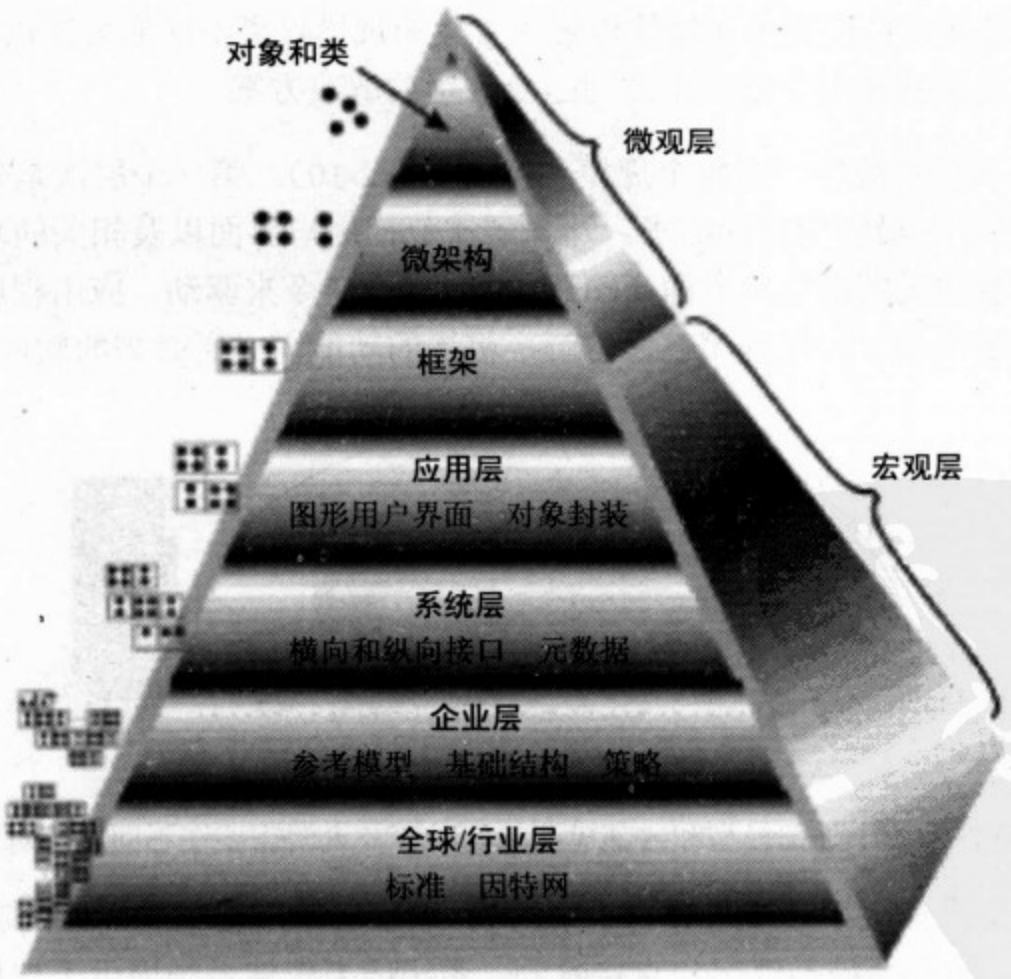



图2-11　软件设计层次模型

36 图2-11显示了架构的7个层，分别是：全球、企业、系统、应用、宏构件、微构件和对象。

① 即《设计模式》（GoF）一书中的模式语言。——编者注

全球层包含的是可以普遍应用于所有系统的设计问题。这一层关心的是所有开发机构之间的协调，涉及跨机构的交流和信息共享。企业层关注的是单个开发机构中的协调和交流。机构可以分布于多个地点，使用不同种类的硬件和软件系统。系统层处理应用程序之间和应用程序集之间的通信和协调。应用层关注的是为满足一组用户需求而开发的应用程序的组织结构。宏构件层关注的是应用框架的组织与开发，而微构件层的核心在于开发软件构件来解决反复出现的软件问题。每个解决方案都是相对独立的，解决的往往是一个更大问题中的一部分。对象层关心的是开发可复用的对象和类。对象层更关心代码复用而不是设计复用。我们将详细讨论每一个层，并给出每个层中所记载的模式的概览。

### 设计复用与代码复用

设计模式强调为开发大规模系统提供可复用的设计指南。设计本身的可复用性对系统的整体开发成本具有显著的影响，远远超过复用个别软件构件所能带来的影响[Mowbray 1995]。为了指出这一点，在图2-12的左侧显示了一个系统，它可以在框架和微架构层利用一些可复用的构件。请注意仍然需要进行系统的总体设计，而整个树的大部分处于设计中的非叶节点部分。由于可以在设计中插入预制的“叶节点”而不是定制它们，减少构件数目确实可以有效地节省时间，但是仍然需要很多的费用来构建整个系统。图2-12右侧显示的系统可以大量复用一个已有系统的设计。虽然系统被限制于以前构建的系统的领域，但是复用的程度要大得多[Yourdon 1993]。通过对设计的复用，与前一个系统相同的所有部分都可以只经过很少的修改就被插入进来。仍然需要针对新用户定制许多的叶节点以满足他们的特定需求。请注意改变叶节点的成本远低于改变更高层设计的成本。这样，优化系统中复用的设计的数量可以为把总体系统成本最小化提供一个框架，它减少的成本远远高于只强调复用个别的构件。

37

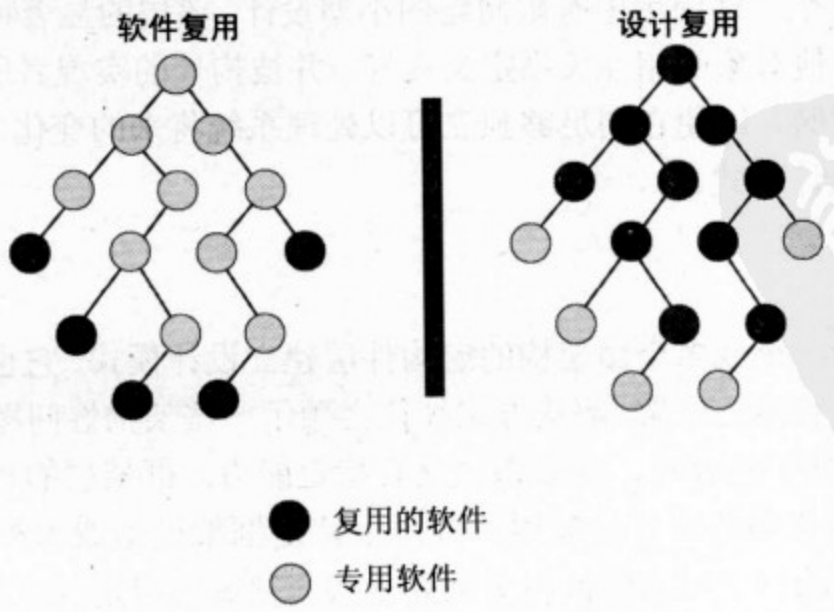



图2-12　软件复用与设计复用

### 2.4.1　对象层

粒度最精细的是对象层。在这里，软件开发人员要考虑对象类和对象实例的定义与管理。在这一层做出的决策包括为每个操作选择特定的对象属性和适当的签名。该层的目标是构建解决应用需求的原始功能性。第二个目标是复用已有的软件和文档，做出在类中包含或不包含哪些内容的决策。一般而言，对象层的讨论是特定而详细的，它们只被用于描述系统实现的细节。

业务层对象可能代表了一个账户或客户的行为和数据。而实现层对象代表的可能是编辑、验
38 证和显示账户或客户数据的一个Java图形用户界面的方法和属性。

在对象层，在类库和编程框架中的对象和类是依赖于编程语言的。有一些方法可以获得语言独立性，例如可以使用OMG（Object Management Group，对象管理组织）的IDL来表示类定义。在CORBA中定义的标准对象模型会定义对象交互的语义。CORBAservice接口还会为管理和控制对象而定义基本的对象层接口。

CORBAservice常被当作积木来构建更大规模的软件构件，这些构件中可能包含有关特定的构件集将如何使用这些服务的策略和规则。依赖于语言的类定义是采用特定编程语言，如C、C++、Smalltalk、ADA、Java或其他语言编写的。依赖于语言的类有可能利用共享的运行时库以便可以被多个应用程序使用。

### 2.4.2　微架构层

微架构层包括那些组合多个对象或对象类的模式。在业务上也许就是一些协同工作的业务对象的集合，用于从信用卡交易记录中计算一个月中的航空里程。而在实现中也许就是一组共同工作的对象，用于为客户或账户对象提供GUI、业务（处理规则）、持久性和查询接口。这一层关心的是开发用于解决一个软件应用中有限问题的小型设计。该层的显著特征在于一组有限的共同工作的对象，它们与其他对象的相互关系定义良好，并被构件的实现者所理解。微架构层模式的目标是复用对构件的封装，以便它们足够独立可以处理系统将来的变化。GPL主要考虑在该层为应用程序开发有效的设计模式。

### 2.4.3　框架层

框架层考虑在包括一个或多个微架构的宏构件层建立设计模式。它也许是用于存储分组的客
39 户及其账户信息的基于容器的框架。解决方案往往会预示一些架构性问题如对象请求代理（object request broker，ORB）架构的存在，或者系统具有特定能力。框架层的目标是允许软件代码的复用以及允许用于编写该代码的设计的复用。专用于特定框架模型或宏构件架构的模式处于这个层。在这一层中，有效的模式可以降低与框架处于共同领域的应用程序的构建成本及维护成本。Taligent公司和西门子公司提出的许多模式和指导意见都处于这一层[Bushmann 1996]。框架试图在被用于解决特定领域的问题时复用大部分的设计和软件。

### 2.4.4　应用层

应用层位于框架层之上。应用中通常包含大量的对象类、多个微架构和一个或多个框架。应用层考虑的是单个应用程序中使用的设计模式。单个开发人员往往会控制一个应用程序（在应用层）如何设计结构、如何受到控制和管理。应用层首要的目标是实现软件需求中定义的特定功能集。它们的功能必须同时满足性能要求。该层包含对结构和设计方法的不同安排。由于它的范围被限制在单个程序，因此该层中涉及的风险远低于在更大规模上（影响到多个应用、系统或企业）涉及的风险。如果正确地设计了更高规模的架构，一个应用中的变化的影响范围就可以被限制在单个应用程序及其所操作的数据上。

应用层包含了实现软件系统的外部模型的程序，即现实世界的运作模型。特别是最终用户的外部需求在应用层得到满足，包括与用户界面和可见的系统功能相关的问题。应用层软件开发活动包括对遗留系统的对象封装和新应用的开发。COTS应用以及各种协作框架都处于这一层次的模型中。

由于其他的著作充分覆盖了更细粒度的层，所以本书的模式语言只关注在应用层和更高层次上发生的问题。迄今为止，在应用层和更高层次的设计模式方面所作的工作非常少，但是面向对象架构恰好在这些层次上是最重要的。



应用模式覆盖了一组不同的解决方案。应用层上的COTS软件和开发支持环境正在迅速地革新。本书的模式语言所选择的应用模式包括库、框架、解释器、事件驱动、持续性以及其他。它构成了应用模式的一个稳健集合，可以解释大多数应用层架构下的原则。

### 2.4.5　系统层

一个系统包括多个集成的应用程序来提供各种功能。系统层在应用之间增加了互用性。系统还负责管理生命周期问题，例如系统的演化。系统层架构是经历系统整个生命周期中对组件应用程序的修改和替换而保持不变的持久结构。例如，它可以是一个集成了海事、家庭和车辆保险应用的保险系统，或者是用于集中控制飞机系统监控、着陆指示系统（Instrument Landing System，ILS）、接近告警和自动驾驶的飞行系统。

系统层很有趣，因为与应用层相比，作用力发生了显著的变化。随着我们移动到更大的规模，变化和复杂性的影响急剧地增加。在一个应用中可能只有不太频繁的变化；而在系统层，这些应用层的变化可能会累积形成系统范围的影响。例如，如果根据一个交错排列的时间表对12个合作的应用程序每年升级一次，整个系统实际上每个月都会升级，带来的风险就是对现有软件的变化可能会影响到其他的应用程序。

每个应用可能都是具有数百个类和数千个方法的复杂程序。随着我们把规模扩大到系统层，系统复杂性的增长速度会高于单独的应用的复杂性增长速度。从某个角度来看，系统类似于由单独的应用结合成的一个大程序。由于应用程序之间需要通信，就需要额外的软件来实现应用之间

的互联。如果未能正确管理这种复杂性，具有*N*个应用的系统就会成为一个具有*N*个复杂模块和*N*×*N*个互联的大得多的程序（见图2-13）。系统层的复杂性还包括通信机制和分布式处理。

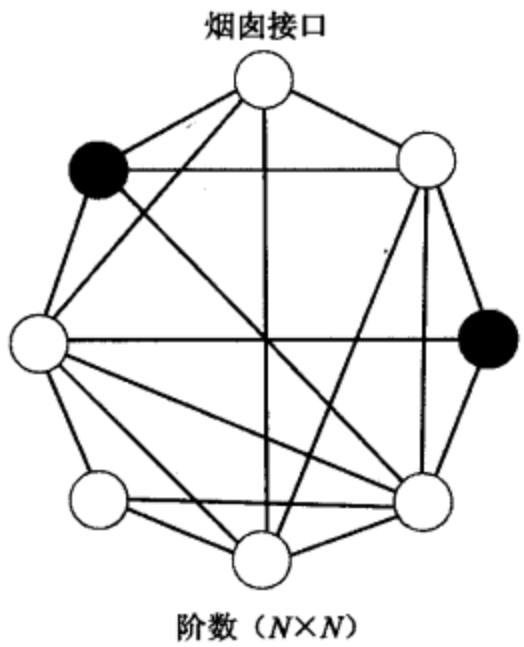



图2-13　如果每个子系统都具有独特的接口，系统就会过于复杂

这种明显的复杂性还会导致在解决方案中出现大范围的差异。很多系统层解决方案都是为了
41 特定的应用实现而专门或惟一定制的。在一个企业中，可能会有许多采用不同架构的软件系统。系统层的目标是提供一个基础结构，让应用程序可以很容易地“插入”进来而获得一定的互用效益。这时的目标就是复用架构以便应用程序可以从系统之间的共同性中获得包括互用性和软件复用方面的好处。

在系统层，变化管理和复杂性管理是最重要的两种原力。而功能和性能管理在应用层更为重要，因为它们在那里受到直接的控制。通过对系统架构进行适度的抽象可以实现复杂性管理。变化管理关注于开发公共接口。它定义了如何对服务进行访问。公共接口允许对应用和系统中的构件进行替换。良好的架构就是把适量的抽象和适当的公共接口集合匹配到一起。

系统层实现了系统的内部模型，并提供应用程序之间有效互用所需的内聚性（见图2-10）。这一层中的三个关键要素包括横向接口、纵向接口和元数据。*横向接口*是为了在机构中的复用而设计的公共接口；例如，横向接口可能包括用于数据传输和访问的通用操作。它们可以为应用提供复用、互用和管理机制。*纵向接口*是考虑到领域特定的需求和纵向力量而定制的接口。向最终
42 用户提供功能和优化性能是纵向接口背后的关键驱动力量。*元数据*是自描述的信息，定义了系统中可用的服务和数据。元数据让系统具有灵活性，具有对变化进行管理的动态能力。横向接口、纵向接口和元数据一起构成了系统层软件架构。

在系统层中，模式被分成结构模式和行为模式两类。*结构模式*具有特定的结构或一组相互关联的构件。系统层结构设计模式包括网关、仓库、构件和特定领域的面向对象架构。*行为模式*定

义了系统在不同情况下的行为表现。行为模式包括各类复用、客户机/服务器、多层和自动化。

### 2.4.6　企业层

企业层是一个机构中最大的架构规模。企业层软件包括多个系统，其中的每个系统都由数个应用构成（见图2-14）。它可能是用于银行业、抵押业和保险业的金融服务，或者是负责售票、记账、信号控制和路轨控制的地铁系统。与全球层不同，机构在企业层可以控制它的资源和采取的策略。企业负责建立整个机构所采用的策略和过程。企业层模式包括一些指导原则，用于制定会对企业软件的结构、风格和未来的成长产生影响的架构决策。这些模式可以帮助定义企业层所需的策略，它们与更低层上的自主决策取得平衡。企业层与全球层的另一个区别在于它的影响被限制在可定义的范围内。企业层的目标是通过在整个机构中使用一组一致的策略和服务，提供对软件的访问，把成本最小化。通过根据设计模式建立机构范围内的过程，可以最小化很多典型的机构方面的企业问题。例如，如果机构建立了要采用特定文件格式或网络标准的策略，就可以避免很多与获取尖端产品的灵活性和产品之间由于缺乏数据转移手段而不兼容有关的问题。

43

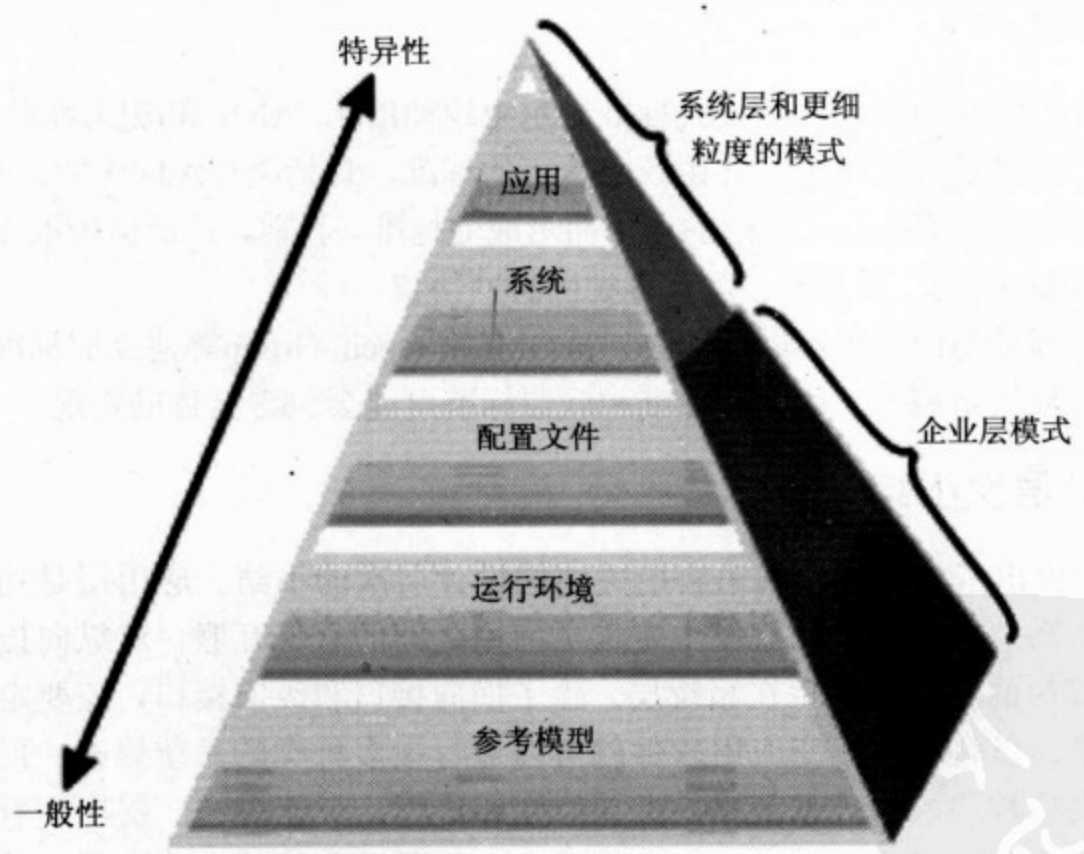



图2-14　企业层架构定义了技术使用策略，包括标准参考模型、公共运行环境和应用配置文件

在企业层要考虑4类软件信息管理：机构的运行环境、分布式进程间通信、资源管理和机构配置文件。此外，在这一层还要针对特定的机构类型做出一定的决策。我们将介绍一些企业层决策的一般特点，作为识别和应用某些不常用模式的指引。

企业层包含了不同的机构组织模型，可以指导一个机构如何利用不同的标准如组织策略、安全和数据访问策略。还包括诸如可用通信协议和共享资源的分布在内的机构基础结构事项。

### 2.4.7　全球层

全球层是最大规模的架构层次，由多个企业构成。定义全球系统的边界很困难，甚至是不可能的，它包括被世界上多数机构广泛采用的事实上的和正式的标准的混合体。解决的关键问题主要涉及软件跨越企业界限的影响。全球层包括语言、标准和影响多个企业的技术策略。全球化系统是那些连接多个企业，而且可以被多个企业联合控制的系统。全球层的目标是多个企业的联合
44 目标。例如，全球层可能会提供一组标准和协议，通过在多个企业间建立某种通用的互用和通信手段而让多个机构受益。

全球系统的最好例子就是因特网，它通过一组世界范围内广泛存在的相关标准和策略进行定义，让信息共享成为可能。因特网的一个重要方面就是各种标准的集合，任何希望共享和访问其他机构中信息的人都可以支持这些标准。对这些标准的使用超越了任何特定机构的控制，对所有想参与的个人和团体都是开放的。

全球层还包括软件标准。在计算机行业中有4类主要的标准：正式标准、法律标准、事实标准的和联盟标准。

（1）正式标准是由经授权的正式标准制定组织如ISO、ANSI和IEEE等倡导的标准。

（2）法律标准是法律要求、并由政府认可的标准，包括类似Ada95和GOSIP的标准。

（3）事实标准是那些由于广泛的应用而形成的标准。目前，广泛应用的事实标准包括微软的Windows和Office标准、TCP/IP、以及各种因特网协议。

（4）联盟标准是由各种不同的团体，如OMG和Open Group等建立的标准。一般来说，正式标准和法律标准只是规范，而事实标准和联盟标准可能会包含具体的实现。

### 2.4.8　设计层次小结

图2-15中给出了本书的模式语言所关注的设计层次的小结。应用层是功能和性能满足用户需求的地方。系统层在更大的规模上定义了应用之间的软件互联。对纵向接口和用于应用之间互用的横向接口的开发也发生在系统层。除了横向接口和纵向接口，还要定义元数据来获得运行时的灵活性，以允许不改变应用软件就可以进行很多种类的系统修改。再大的规模就是企业
45 层，在此定义机构层次的技术策略、指导原则和过程。企业在这一层控制着参考模型和运行环境，做出选择来延伸机构的需求。在全球层中，将通过和其他企业取得一致，达成技术协议来建立标准。


图2-15　软件设计层次模型显示了第5章~第7章中的反模式所针对的层次

## 2.5　架构规模和原力

原力是对大部分软件决策有重大影响的关注点。在这些关注点之间做出的折中会影响到所有层次的软件架构的总体质量。为了简化决策的制定，澄清对问题的隔离，我们估计了各种原力在不同架构规模上的相对重要性（见图2-16）。

46

| | 应用层<br>“程序员” | 系统层<br>“架构师” | 企业层<br>“信息总监” | 全球层<br>“执行总监” |
|---|---|---|---|---|
| 功能管理 | 1 | | | |
| 性能管理 | 2 | | | |
| 复杂性管理 | | 2 | | |
| 变化管理 | | 1 | 2 | |
| 资源管理 | | | 1 | 2 |
| 技术转移管理 | | | | 1 |

图2-16　原力在每个架构规模上的相对重要性是变化的

在应用层，软件开发人员在开发软件以满足用户的功能要求时主要关心功能管理和性能管理。系统层关注的是为了减少开发和维护机构中软件系统的整个生命周期成本，开发可以有效管理复杂性和变化的基础结构。系统层架构师负责提供可适性，保证总体系统设计足以适应机构的发展需要。

企业层的重点是资源管理，因为大规模系统常常涉及数千个相互作用的构件和服务。此外，变化管理的重要性也超过了其他在较低层次上得以解决的原力。随着企业规模的增长，资源管理成为了更主要的关注点。

在全球层，重点是支持现有的标准，有效地利用已有的知识库来增强企业层开发。不过，由于机构对外部技术的影响是有限的，企业对解决方案的控制能力有所下降。在新技术和IT资源利用上保持领先是全球层的主要考虑。

本书的模式语言将考查每个设计层次上反复出现的结构，为每个架构规模上固有的主要问题提供详细的解决方案。设计模式擅长建立平衡各种原力的可复用的架构设计。有效平衡了各种力量的良好面向对象架构有一个自然的作用，就是可以减少总体软件开发成本和维护成本，而且在生命周期的大部分时间都可以使用软件所提供的服务。

47
~
48

# 第3章 模式和反模式的模板

对模板的使用让设计模式和反模式不同于其他形式的技术讨论。模板保证了在模式语言、模式分类或模式系统中对每个模式都回答了那些重要的问题。如果没有模板，模式就只不过是某个人未经组织的文字，或者是某人说："这是一个模式，因为我是专家而且我说它是个模式。"我们不会接受这种论据，而你也不应该接受。没有结构就很难确定什么是模式而什么不是模式。因为如果那样，在普通技术讨论和模式文献之间就失去了可以识别的形式或区别。

模式语言（或模式分类）是一组相互关联的模式的集合。每种模式都采用一致的修辞结构——模式模板。我们使用这种修辞结构来表示对模式的说明是具有很好逻辑的。这种一致的、合乎逻辑的结构是使用模板的直接结果。模板的每个部分都具有修辞性的目的。每个部分都是用于完成推理的一部分，而且都回答了与涉及的模式有关的一些关键问题。

修辞结构的意图和范围在不同模式分类中是有所区别的。模板的结构经过选择，以符合读者的需要和作者的资源。接下来几节将说明一些主要的模板形式。



## 3.1 退化形式

退化模式（degenerate pattern）没有可以辨别的模板。与模板化模式（templated pattern）相比，退化模式只包含一个单独的内容部分。下面是退化模式形式的一个例子。

### 退化模式形式

**技术讨论：**作者对这组概念想表达的某个观点

这是一个模式吗？这是个很主观的判断。大家可以称它为一个模式，因为它的作者选择这样做。我们也可以仅仅只是因为作者的名气不够大或者其他个人原因而不把它当作一个模式。

退化形式无法提供实践者可以依赖的特定惯例或指引。读者必须分析作者的松散文字来区分哪些内容分别是问题、意图、上下文环境、作用力、解决方案、益处、后果和例子，等等。实际上，很可能会缺少其中的某些元素。

## 3.2　Alexander 形式

Christopher Alexander的传统模板由3部分组成：名称、问题和解决方案。“因此”这个词把讨论划分成不同部分。“因此”出现在对问题的讨论和对设计方法的讨论之间，下面是Alexander模式形式的一个例子。

### Alexander 模式形式

**名称**：作者如何称呼这个概念？
**技术讨论**：有哪些相关的问题？

**因此**

50　解决这些问题的常见解决方案是什么？

Alexander形式包含了对动机和采取的行动的基本划分。按照惯例，Alexander形式中会包含一个图。它是对文字中表述的思想的空间抽象。Alexander形式是对任何设计模式进行说明时可以得到的最小形式。

## 3.3　最小化模板（微型模式）

软件工程设计模板的最小化修辞结构包括名称、问题和解决方案。名称定义了独特的术语；问题列出上下文环境、作用力和/或适用性；而解决方案则是模式的内容（为了解决问题需要做什么）。这种结构适合于低开销的模式文档化工作，例如研讨会上的模式开发或现场咨询。这时方便和满意是最关键的，而并不需要什么正式形式。它也是一种简明的方式，某些出版物对此有所要求。下面是微型模式的标准模板。

### 微型模式模板

**名称**：实践者应如何称呼这个模式？
**问题**：驱动我们应用该模式的因素是什么？
**解决方案**：我们如何解决该问题？

## 3.4　小型模式模板

小型模式对问题或解决方案进行了分解，揭示出有指导意义的元素。模式中接下来最重要的修辞方面就是那些“指导”元素。尤其是上下文环境、作用力和/或带来的益处或后果。最重要的是，这些部分回答了解决方案中有关“为什么”的问题，同时还提供了关于何时使用该模式的信息。下面有两个小型模式模板的例子。

### 3.4.1　归纳式小型模式

**名称**：实践者应如何称呼这个模式？
**上下文环境**：应用这个模式时设想的环境或先验假设是什么？ 51
**作用力**：要平衡哪些不同的设计动机？
**解决方案**：我们如何解决该问题？

### 3.4.2　演绎式小型模式

**名称**：实践者应如何称呼这个模式？
**问题**：什么原因驱使我们应用这个模式？
**解决方案**：我们如何解决该问题？
**益处**：应用此模式的潜在正面效果是什么？
**后果**：应用此模式的潜在缺点和负面后果是什么？

我们为小型模式说明了两个模板形式，分别是归纳式和演绎式。归纳式小型模式强调模式的适用性，而演绎式小型模式则强调解决方案的结果，即带来的益处和负面后果。

## 3.5　正式模板

在详细编写模式和反模式时还可以增加很多部分。包括对其他相关模式和非模式技术概念参考在内的引用对完整、正式的处理方法也是有益而且必需的。下面是对在出版物中使用的一些正式模板的归纳。请注意在部分名称后面的文字是对作者最初的模板定义的解释性概括。GoF模板是适用于微观架构层模式的正式模板[Gamma 1994]。

### 3.5.1　GoF 模板

**模式名称**：如何称呼该模式？
**模式分类**：该模式是创建模式、结构模式还是行为模式？
**意图**：这个模式解决什么问题？
**别名**：这个模式还有哪些其他名字？
**动机**：应用这个模式的示例场景是什么？ 52
**适用性**：何时可以应用这个模式？
**结构**：这个模式中对象的类层次图是什么样的？
**参与者**：有哪些对象参与到这一模式中？
**协作**：这些对象如何互用？
**效果**：使用这个模式时做出了哪些平衡？
**实现**：应用这个模式时有哪些技巧或问题？
**代码示例**：这个模式的源代码例子是什么样的？

**已知应用**：有哪些使用这个模式的真实系统？
**相关模式**：这个模式集合中的其他模式与这个模式的关系如何？

### 3.5.2　模式系统模板

模式系统模板是适用于多个模式层次，如惯用法层、应用层和系统层的正式模板[Buschmann 1996]。

**模式名称**：如何称呼该模式？
**别名**：这个模式还有哪些其他名字？
**示例**：需要这个模式的例子是什么？
**上下文环境**：何时应用这个模式？
**问题**：这个模式解决什么问题？
**解决方案**：这个模式下潜在的根本原则是什么？
**结构**：这个模式包括和涉及哪些对象（结构图）？
**动态**：这些对象如何协作（交互图）？
**实现**：实现这个模式的指导原则有什么？
**解决的示例**：显示如何使用这个模式来解决前述的例子。
**变化**：这个模式有哪些重要的变化形式？
**已知应用**：使用这个模式的真实系统有哪些？
**效果**：使用该模式的好处和坏处分别是什么？
53　**参见**：有哪些相关的模式，它们的区别是什么？

## 3.6　对设计模式模板的反思

模式的早期读者和实践者使用正式模板时的大部分实际经历并不令人满意。模式传递的通常是相当简单的概念。但是，许多读者发现模式的说明过于关心冗长的细节、相关问题和先期讨论。考虑到模式模板从高深莫测的“退化模板”开始已经经历了很长的发展时间，读者的这种反应引人注意。

大家需要的是一种迅速、便捷的信息传递方式。我们使用的一种关键方法就是利用建立于稳健的参考模型上的模板。*CORBA Design Patterns*中的参考模型允许我们定义出现于不同软件设计规模上的横向（共同的）上下文背景和作用力，而不是让所有模式都完全相互独立。通过单独一章来说明参考模型，处理这种共同性，我们可以对几乎每个模式都消除数页纸的冗长先期讨论。

此外，我们希望模板的结构易于理解。考虑到大多数实践者要学习和吸收很多的模板，而他们的工作时间都很紧张，我们把模板的各部分组织成让读者可以先了解一个模式的大部分内容，再接触到散漫的、自由的文字讨论。

CORBA设计模式模板的组织形式是把提供大量最简明信息的部分放在最前，而自由形式的部分放在最后。该模板开始于在参考模型中定义的关键字，接下来是方案意图的摘要和解决方案的抽

象图。适用性是按照重要性排序的符号列表，简明地覆盖使用该模式时最重要的动机。解决方案摘要清楚地解释了该模式的技术途径。跟在解决方案摘要后面的各部分是使得该模式可行的依据。

实际上，我们发现在解决方案部分中对各种变化的讨论其实会让一些读者感到迷惑。所以我们专门用一个独立的部分来说明解决方案的变化形式，以便在对解决方案的解释中不会出现让人疑惑的矛盾态度或警告。另一个部分覆盖了可选的软件规模，说明了模式在应用于不同的软件设计层次时的差异。

### CORBA 设计模式模板

**解决方案名称：**
**解决方案类型：**它是软件模式、技术模式、过程模式还是角色模式（只使用关键词）？ 54
**意图：**该模式是关于什么的（在25字以内说明）？
**原力：**该模式最好地解决了参考模型中的哪些横向力量？
**在此规模的适用性：**该模式何时适用？模式背后的主要动机是什么（有序符号列表）？
**解决方案摘要：**模式的解决方案和途径是什么（清晰说明，不要使用冗长的修饰）？
**效益：**应用此模式的主要正面效果是什么（符号列表）？
**其他后果：**应用此模式的主要负面后果是什么（符号列表）？
**改变解决方案规模到其他层次：**该模式如果应用于其他软件规模层次会有什么不同？
**相关解决方案：**到其他模式（包括来自其他模式出版物的模式）、其他技术示例、出版物引用和资源的交叉引用和引用。基本上是指向在当前模式文本之外可能会对读者有用的所有内容。
**示例：**该模式在应用背景中的实际例子，如果可以的话包括源代码列表。
**背景：**其他相关信息。

## 3.7　反模式模板

反模式是一种新形式的模式。模式和反模式间的基本区别在于反模式中的解决方案产生的是负面后果。某些结果也许当前就很明显（症状），某些则是可以预计的（后果）。反模式具有一些基本的形式。

反模式要有用，就要再包含一个解决问题的方案。因此区别模式和反模式的另一个方法就是在反模式中有两个解决方案（而不是一个问题和一个解决方案）。第一个解决方案产生负面后果（必需被解决的作用力）。而第二个解决方案则是对第一个方案的迁移（或重构），可以提供显著改善的效益，而不良后果则大大减轻。

在普通设计模式中，惯例是至少必须已知3个对该解决方案的应用。由于在反模式中有两个 55
解决方案，反模式的“三法原则”与模式不一样。由于没有3个解决方案会是完全一样的，所以模式解决方案是对已知应用中最佳实践的抽象。

反模式的不同在于它们有两个解决方案。第一个解决方案是“反模式”，必须和普通模式一

样遵守出现3次的原则。不幸的是，完全相同的解决方案不太可能适用于反模式所有已知的发生情况。如果问题存在多个解决方案，反模式中的第二个解决方案就是根据该反模式的已知解决方案得到的最佳实践方案。在模板中同样可以对这些解决方案加以说明。

### 3.7.1　伪反模式模板

反模式有一种退化模板形式，作者使用贬义语言描述一个不好的解决方案。因特网上某些最早出现的反模式采用的就是这种形式。这并不是一种特别有用的形式，因为它所采用的轻蔑态度过于主观和片面。

**名称**：如何称呼该反模式？
**问题**：它受到贬斥的特点是什么？

### 3.7.2　小型反模式

正如我们讨论过的，反模式模板与模式的区别在于前者介绍了两个解决方案。第一个解决方案被称为*反模式问题*，而第二个方案被称为*重构方案*。可以使用其他合适的模板部分来扩充这个最小形式的模板。

**名称**：实践者应该如何称呼这个反模式（贬义的）？
**反模式问题**：重复出现的导致负面后果的解决方案是什么？
**重构方案**：我们如何避免、最小化或重构该反模式问题？

小型反模式可以带有对如何解决该反模式的讨论，以及以补充内容的形式包含一幅卡通画或
56 轶事。它的目的与微型模式和小型模式相似，都是以不太正式的方式提供简练的模式说明。

## 3.8　完整的反模式模板

本书剩余部分将使用下面这个反模式模板来完整地记录反模式。书中还以补充材料的形式用被称为小型反模式的最小化格式（前面介绍的模板）提供了一些较简单的反模式。完整的反模式模板包含一些必需的和可选的部分。反模式的*一般形式*和*重构方案*是最核心的部分。它们提供了组成一个反模式的问题方案/解决方案对。

- *反模式名称*。反模式名称是一个惟一的名词词组。它被故意取为带有贬义。模式的解决方案名称成为了新的术语。该反模式中所包含的原则使用这个名称来引用这个方案。重要的是让每个反模式都有名称，以便惟一地确定它。名称的关键作用在于它们构成了一个机构中的成员对软件和架构进行讨论和记录时使用的术语基础。
- *别名*。该部分说明用于该模式的其他常用的或说明性的名称和短语。
- *最常见规模*。使用参考模型中的一个关键字来说明规模。这一部分确定了反模式适用于软件设计层次模型（见图2-7）的位置。每个反模式都被放置在它在逻辑上最适用的位置。对位置的次要考虑则是作为结果产生的重构方案的规模。可以从惯用语层（对象层）、微

架构层、框架层、应用层、系统层、企业层和全球/行业层中进行选择。规模还说明了解决方案的尺度。某些模式在多个规模上定义了有用的解决方案。在反模式模板中也说明了解决方案在不同规模上的形式。

- 重构方案名称。该部分是重构方案的标识。
- 重构方案类型。该部分包含来自参考模型的关键字。这些来自软件设计层次模型的关键字指明了该反模式的解决方案所引发行动的类型。你可以选择的类型包括软件、技术、过程和角色。“软件”说明解决方案将产生新的软件。“技术”说明需要获取新的技术或产品。“过程”指明解决方案要求采用某个过程。而“角色”说明需要让个人或群组承担某些职责。下面是这4种不同解决方案类型的详细说明：
  - 软件模式囊括了反模式分类中的绝大部分模式。软件模式要求建立新软件。计算机行业中现有的绝大部分设计模式都是软件模式。
  - 技术模式通过采用某种技术，例如Java来解决软件问题，而不是从头开始编程实现需要的功能。技术模式也是设计模式的原因在于，虽然获取它们的方法不同，但它们也会引起软件设计和实现。技术模式可能也包含一些编程，例如为商业软件模块建立一个对象封装器。
  - 过程模式提供对某项活动的定义，对特定解决方案可以一致地重复进行该活动。
  - 角色模式通过给开发机构的利益相关者分配清晰的责任来解决软件问题。包含过程模式和角色模式的原因在于交流和人员组织本身对软件问题的解决都具有重大的影响。有些时候，我们发现简单的过程或对职责的澄清为解决技术问题提供了最有效的行动杠杆。
- 根源。这一部分包含来自参考模型的关键字。它们是导致该反模式的一般原因。请从来自第2章有关根源的这些关键字中进行选择：匆忙、漠然、思想狭隘、懒惰、贪婪、无知、自负或责任（通用原因）。
- 不平衡的力量。该部分包含来自参考模型的关键字，指出在该反模式中被忽视、误用或过度使用的原力。可选的内容包括功能管理、性能管理、复杂性管理、变化管理、IT资源管理和技术转移管理。风险是所有这些力量中的内在作用力。请注意IT资源管理包括对一般资源（包括经费资源）的跟踪和分配问题。
- 轶事证据。这是可选的部分。出现在这部分的是那些与该反模式有关的大家经常听到的说法和喜剧性的材料。
- 背景。这是可选的部分。背景中可以包括更多出现问题的例子，或者有用或有趣的一般背景信息。
- 该反模式的一般形式。该部分常常有一张图，用来说明该反模式的一般特点。它不是一个例子，而是一般化的说明。通过一段文字对该图（如果有的话）进行解释，给出该反模式的一般说明。重构方案将解决此部分提出的一般反模式。
- 症状和后果。该反模式导致的症状和后果的符号列表。
- 典型原因。该部分的符号列表指出了导致该反模式的独特原因（用于补充前面已经指出

57
58

的根源）。

- 已知例外。反模式行为和过程并不总是错误的，它们往往在某些特定场合下是适当的做法。该部分简要说明每个完整反模式的主要例外。
- 重构方案。该部分说明了一个重构过的解决方案，用于解决在该反模式一般形式部分中说明的作用力。对该解决方案的说明不带有它的变化形式；在变化部分中才对它们加以说明。对该方案的说明结构是按照方案的步骤来组织的。
- 变化。这是一个可选的部分，列出了该反模式所有已知的主要变化形式。此外，如果有可选的解决方案，同样在此加以说明。变化部分之所以存在的部分原因是为了增强一般形式和重构方案部分的清晰性。在没有有关主要可选项和替代设计点的警告的影响时，可以更清晰地说明解决方案。变化部分包含了这些扩展，延伸了该解决方案的能力。
- 示例。示例通过列出解决方案的细节来显示如何将该解决方案应用于对应的问题。这一部分通常包括下列元素：问题图、问题说明、解决方案图和解决方案说明。该部分包含从经验中提炼出的一个或多个该反模式的例子。
- 相关方案。该部分指出所有合适的引用或交叉引用。在这里列出所有与当前反模式相关的反模式，并解释它们之间的区别。该反模式与其他模式语言中设计模式的关系也在此加以指出和说明。对相关模式的参考是反模式的一个重要方面。每种反模式都会解决一些力量，同时产生一些新的力量。通过使用相关的模式有可能在同一个层次，也可能在不同的层次解决这些新的力量。该部分还重点说明了相似模式之间的区别。

59

这一部分还包含相关的术语、参考和资源。在这里解释相关术语的目的有两个：区分定义与使用相似名称的术语，以及联系不同名称指向的相关概念。这两种模糊性导致了软件过程圈子中的许多混乱。参考中包含众所周知的术语、示例技术和相关的研究。这些参考对专家尤其有用，他们可以使用这些信息迅速地把该模式与其他已知的著作联系起来。如果一个专家级读者全面理解了一个或多个这些参考，那么他就早已用一个不同的术语了解到该模式的核心思想（我们在使用其他模式语言的时候遇到过这种效应。如果没有这一部分的帮助，有时需要花大量的时间才能消除这些术语差异对理解的影响）。这一部分既是参考列表，也是关于其他著作的“别名”同义词列表。资源中包括指向针对该问题的其他类型信息和机构的指针。

- 对其他视角和规模的适用性。这一部分定义该反模式如何影响其他的视角：管理视角、架构视角或开发人员视角。它还说明了该模式与其他开发层次的相关性。如果模式在不同层次上采用了不同的“名称”，就应该在此说明。在此部分解决的某些关键问题包括：该反模式应用于不同层次时会怎么样？它解决其他规模上的作用力时有效性如何？不同的层次会引起哪些新力量，以及它们是否也获得了解决？影响到该设计模式的关键力量如何随规模而变化？关键设计元素的角色如何随规模而变化？

第5章~第7章中的反模式使用这个模板来记录软件行业中常见的功能不良的实践，并提出至少在3个场合中被证明有效的实际解决方案。这些章讨论了软件开发反模式、架构性反模式和管理性反模式。我们选择这些层次来完整覆盖软件项目所涉及的关键问题。

60

# 第4章 对使用反模式的建议

使用反模式来变革机构中已有的实践时存在一种危险。某些在早期采用反模式的人以破坏性的方式使用它们。虽然他们在自己所在的机构中发现了某些与本书中讨论的反模式相似之处，但是这并没有成为变革的催化剂，而是导致了责任人提前退休或者被发配到公司在边远地区的部门。虽然指出过失者和指责他们也许会在短时间内提供一种满足感，但这绝不是我们期待反模式应发挥的作用。

重要的是记住每个公司在任何时候都可能会受到数个反模式的影响。但是，没有出现反模式并不能保证一个机构一定会成功。事实上，许多机构虽然反复违反一些常识——出现了反模式——却仍然取得了成功。毫无疑问，解决反模式会引向软件的改进，但并不是需要解决所有反模式才能取得成功。反模式最适合解决长期问题，尤其是为了达到机构的目标必须先期处理它们时。请听从这个建议："如果东西还没坏，就不要去修它；让它就保持原样。"



反模式的目标并不是强调机能不良的软件开发实践，而是提出和实现一些策略来修复出现的问题。对软件项目的改进最好通过增量的方式来完成。一个要求同时纠正多个反模式的计划是高风险而且不明智的。通过按照仔细考虑的计划逐个问题改进开发过程，可以显著提高成功实现一个解决方案的可能性。而且，解决方案的实现只有在技术人员具备解决问题所需的技能时才是可行的。当具备的技术知识不足以完整实现反模式解决方案时，这个治疗过程可能引起和原始问题一样糟甚至更糟的问题。

## 4.1 机能不良环境

软件行业中的机能不良环境就是对机构策略的讨论出现争论，而消极和负面意见主导了技术讨论的环境。我们在一些软件开发机构中经历过这种环境并挺了过来。我们讨厌工作在机能不良的环境中，你也不例外。

反模式并不是要恶化机能不良的工作环境。正好相反，环境越机能不良，技术讨论就越肤浅、越无趣，产出也越低。更为深刻的技术讨论对软件工程的发展是不可缺少的。反模式的意图是建

立对问题的警觉。使用反模式，你可以在机能不良的实践导致负面后果时清楚地看到问题。反模式给有害的实践赋以鲜明的名称和特点。让你可以迅速检测到它们的存在，然后避免或解决它们，再转向更加关心的和更有产出的问题。

反模式不会完全改变你的工作方式。你可能在软件工程事业生涯上已经相当成功，但是反模式仍然可以产生新的警觉。它们是帮助你穿越软件行业中的困难环境及其内在矛盾的智力工具。

## 4.2　反模式与变化

62 显然，变化在软件开发中是必需的。当5/6的公司软件项目不能成功时，一定有什么地方出了严重的错误[Johnson 1995]。而在机构中出现差错的时候，原因几乎总是一样的：有人没有承担起职责[Block 1981]。因此，最重要的变化就是激励大家勇于承担解决问题的责任。要帮助大家认识到自己的职责，就需要进行某种形式的干预。

一个典型的干预模式被称为“剥洋葱”。这是一种包含了3个问题（按下面的顺序）的谈话方式：

（1）问题是什么？
（2）其他人为解决此问题做出了哪些贡献？
（3）你为解决此问题做出了哪些贡献？

这个方法被用于在机构中收集信息。回答这些问题可以建立起对个人职责的警觉。

干预的第一步是让参与者承认存在问题。这被称为提高觉悟。软件反模式清晰地定义了最常见的问题，而对它们的症状和后果的说明则提供了检测问题和预见其结果的办法。

对替代方法的了解是解决问题的关键之一。乐观主义者会告诉你总会有替代方法，而且永远、永远、永远不会没有。了解到有很多方法可以替代有问题的实践，对实现变化是一个重要的促进。换句话说，本书推荐的反模式解决方案只是很多可能的替代方法中的一部分。某些反模式的解决方案甚至可能也适用于其他反模式。因此，即使某个反模式的一般形式可能第一眼看上去并不适用，阅读所有的解决方案仍然是有益的。

### 固定反应和即兴反应

模式和反模式记录了一些已知的替代解决方案。对刺激实际上只有两类反应：即兴反应和固定反应。即兴反应是指在受到刺激的时候临时拼凑出某种做法。这不是一个模式。而固定反应则是某种习得的或实践过的做法。它是一个模式或反模式。该反应会产生正面的还是负面的后果取决于模式的上下文环境和执行过程。如果上下文环境导致了负面后果，它就是反模式。如果环境
63 带来的是正面效果，它就是模式。

如果模式被应用于错误的上下文环境，就可能会变成反模式。在技术领域，这种情况以过时

术语的形式一直在发生，例如“结构化编程”和“Altair 8080”。随着技术范例（模式）存在的时间越来越久，它的缺点逐渐广为人知。最后，这个技术范例会被大多数实践者看作一个反模式。这个称呼本身就为它被新技术的取代提供了重要的解释。

## 4.3　编写新反模式

本书建立了反模式的研究领域，但它并没有定义所有可能的软件反模式。第5章~第7章的分析虽然相当全面，但也只是对软件开发文化中的反模式的管中窥豹。令人吃惊的是，一些有经验的人员却从未听说过最基本的反模式（Spaghetti Code和Stovepipe System）。本书将解决这种不足。

软件文化充满了孤立的发展过程，还有相当多的内容需要学习。反模式是捕获知识、传递思想和促进交流的有效方法。让整个行业中的软件开发实践者都来辨识与记录反模式并写下已知的解决方案是相当重要的。对反模式的文字记录对于把信息传递给其他人，并让行业中就机能不良的实践是如何产生的以及应如何有效地加以解决达成一致意见，是非常关键的。有一些重要的观察结果可以帮助你将来在模式和反模式方面的工作：

（1）*设计模式开始于一个多次出现的方案*。设计模式通常是自底向上编写的；也就是说，设计模式开始于一个多次出现的方案。然后该模式的编写者会添加上下文背景和影响力量。用来说明这些背景和力量的文字经过仔细地修改，以便引导你得到惟一的解决方案，也就是一开始的那个解决方案。这就是很多设计模式都难以阅读的原因。作为设计模式的读者，你的第一个任务就是破解隐藏在这些背景和力量背后的逻辑关系。这个逻辑关系往往会引导你达到作者预先确定的结论，而不是得到对实际情况中的作用力、症状和后果的真实理解。解决方案看起来常常是明显而且简单的，尤其是在经过了详细深入的分析之后。反模式则试图以更为吸引读者的形式来讨论适用于设计模式的素材。

64

（2）*反模式开始于一个多次出现的问题*。反模式是自顶向下编写的。反模式开始于一个多次出现的设计或实践。在当前上下文环境中，该设计或实践通常具有显著的负面后果。选择一个或多个贬义的名称，最好是被广泛使用的名称来称呼这个反模式。反模式的说明中包含诸如症状、后果和原因等学习材料。我们选择一起列出症状和后果。症状是过去的和现在的表现；而后果则是将来的。把症状和后果区分开可以为反模式的状态建立起时间期限。

（3）*采用研讨组*。在团组环境中对反模式及其重构方案的文字说明进行讨论是有益的；作者研讨会是其中一种形式。我们还发现由软件人员组成的不太正规的讨论组，如读书俱乐部也是有效的讨论形式。讨论组可以通过回忆该反模式出现的实例来告诉你写下的这些内容是否确实是一个反模式。

我们采用讨论组回顾了许多设计模式和反模式。从这些经验中，我们发现反模式与普通模式相比有趣得多！一个好的反模式会让人参加到讨论中。好的反模式往往会鼓励大家贡献自己的经验和想法。

（4）*成为反模式作者*。新的反模式作者可以关注于在多个不同的上下文环境中（至少3个，

类似于说明软件设计模式的基础）多次出现的问题。要完整说明一个反模式，重要的是用根源、症状以及具体示例来补充反模式的问题说明和解决方案，以提供一个清晰的上下文环境。在解决方案中，其中一个示例应该讨论使用解决方案中给出的指导原则来成功地在某个场景下进行重构。它并不一定要来自当前的项目，而是可以使用其他项目或其他公司的经验，甚至它可能与当前项目没有直接的关系。重构方案并不需要是惟一的方案，但是它应该与当前问题高度相关。

65 在每个反模式的“变化”部分都包含了对相关反模式和替代解决方案的说明。在*CORBA Design Patterns*[Mowbray 1997c]中，我们发现把变化隔离开有助于让解决方案更清晰。这样，介绍重构方案的时候就不用带着那些会引起混乱的警告。

## 4.4 小结

软件反模式是一个派生于设计模式的新研究领域。刚开始进行这项研究的时候，许多人以为软件反模式会很难确定。但是反模式在软件行业中早已非常普遍。实际上，软件行业从可编程计算机的发明之日起就已经开始建立和使用反模式。

我们相信与普通设计模式相比，反模式是更有效的软件知识交流方法，原因在于：

- 反模式通过明确导致机能不良的软件开发过程的症状和后果，为软件开发人员、架构师和管理者澄清了问题。
- 反模式表达了进行改变的动机以及重构不良过程的需求。
- 反模式对理解大多数软件开发人员所面对的共同问题是不可缺少的。学习其他开发人员的经验和教训是有价值的，也是必需的。认识不到这一点，反模式就还会继续存在。

66

# *Part 2*

# 第二部分

# 反 模 式

本部分内容

# 第5章 软件开发性反模式

我们首次见到软件开发性反模式[①]是在Mike Akroyd的报告[Akroyd 1996]中，他是摩托罗拉公司和其他一些大型公司的软件实验咨询顾问。Akroyd反模式定义了面向对象软件设计中的经典问题。本章中的部分开发性反模式是对他的概念的延伸。所有Akroyd反模式都有一个吸引人的特点，就是它们都包含了一个重构方案。通过结合这个重构方案，让反模式具备了颇有价值的用途：它不仅指出问题，还告诉你应该如何加以解决。

正确的反模式定义了从负面解决方案到正面解决方案的迁移（或重构）。只描述了负面解决方案的反模式被称为伪反模式。可以把伪反模式看做是通过电子邮件进行的激烈抨击。在使用过Akroyd反模式之后，我们在因特网上发现了反模式和伪反模式的例子。PLoP会议也讨论了一些与反模式相关的论文，例如*Big Ball of Mud* [Foote 1997]。



## 5.1 软件重构

开发性反模式的关键目标是描述有用的软件重构形式。软件重构是一种代码修改形式，用于改善软件结构以支持后续的扩充和长期维护。在大多数情况下，其目标是在不影响正确性的条件下转换代码。

良好的软件结构对系统的扩充和维护是必要的。软件开发是一项无序的活动，因此系统的实现结构往往会偏离通过架构、分析和设计而规划出的结构。软件重构是改善软件结构的有效方法。生成的结构并不一定要与最初规划的结构一样。结构发生变化的原因在于，程序员在重构过程中了解到一些改变了编码实现方案上下文背景的限制和方法。正确地使用重构是编程过程中的一项相当自然的活动。例如，本章后面讨论的Spaghetti Code反模式的解决方案定义的软件开发过程就结合了重构。

我们强烈建议先重构，再进行优化。优化常常要求在软件结构上做出折中。理想情况下，优

① 这里“软件开发性反模式”中的“软件开发”实际上相当于我们通常所说的程序设计，主要针对开发人员，请读者留意。——编者注

化只影响程序中的一小部分。先进行重构有助于把优化代码与软件主体分割开。

### 正规重构变换

*正规重构变换*包括超类抽象、条件消除和聚合抽象。这些正规重构起源于Opdyke的博士论文[Opdyke 1992]。它们被称为正规重构的原因在于可以证明它们的实现不会影响程序的正确性。把这些变换自动化也是可行的。下面的重构涉及对对象类、实现和关系的改变。

- **超类抽象**。这种重构应用于两个或更多相似的类。超类抽象建立一个抽象类来合并数个具体类中的共同实现。要进行超类抽象，对程序的转换包括：（1）把相似的方法签名转换成通用方法签名；（2）建立一个抽象超类；（3）修改代码以合并选择的实现；（4）把通用方法迁移到抽象超类。
- **条件消除**。在类的结构和行为严重依赖于某个条件语句时可以应用该重构。主要步骤是：（1）根据每个条件建立相应的新子类；（2）把操作代码从条件类迁移到新的子类；（3）调整对类的引用指向适当的子类。最后这步变化可能会影响到构造函数、类型声明以及要求调用重载的方法。修改过的引用应该维持条件类的原始逻辑状态，作为新类中的不变性断言。
- **聚合抽象**。聚合抽象通过重新组织类的关系来改善它们的结构和可扩展性。这种变换可以采用多种形式：（1）把继承关系转换成聚合关系；（2）迁移聚合类到构件关系；或者（3）迁移构件关系到聚合关系。

68

这三种粗粒度的重构依赖于数十个几乎所有程序员都熟悉的细粒度程序变换[Opdyke 1992]。这些细粒度变换的例子包括：（1）重命名类、方法和属性；（2）建立新类；（3）在类之间迁移功能；（4）把直接引用转换成间接指针，等等。要获得完整的列表，参阅Opdyke的论文[Opdyke 1992]。

## 5.2　开发性反模式摘要

开发性反模式利用多种正规和非正规的重构方法。下面的摘要是对本章中介绍的开发性反模式的概述，主要针对开发中的反模式问题。它包括对开发性反模式和小型反模式的说明。摘要后面的每个反模式模板中介绍了相应的重构方案。

**The Blob（胖球）**：过程式风格的设计导致一个对象承担大量职责，而大部分其他对象则只用于保存数据或执行简单的处理。解决方案包括重构设计来更均匀地分布职责以及隔离变化的影响。

69

**Continuous Obsolescence（持续过时）**：技术变化非常迅速，以至于开发人员往往难以跟上软件的当前版本，无法找到可以共同工作的产品发布的组合。考虑到每个商业产品线都要通过新版本来进行发展，开发人员要面对的情况就越发艰难。找到可以成功互操作的兼容产品发布则更加困难。

**Lava Flow（岩浆流）**：死代码和被遗忘的设计信息会冻结在不断变化的设计中。这类似于在

岩浆流中逐渐硬化的小块岩石。重构方案包括了配置管理过程，可以消除死代码，对设计进行发展或重构来得到更高的质量。

**Ambiguous Viewpoint（模糊视角）**：面向对象分析和设计（OOA&D）模型常常没有澄清该模型所表达的视角。默认情况下，OOA&D模型采用了实现的角度，而这个视角在很多时候很可能是最没有用的。混合的视角使得在接口和实现细节之间无法进行基本的隔离，而这种隔离本应是面向对象范型带来的首要益处。

**Functional Decomposition（功能分解）**：该反模式是有经验的非面向对象开发人员使用面向对象语言来实现应用程序时的产物。生成的代码在类结构上类似于结构化编程语言（如Pascal和FORTRAN）。这些机灵的过程式开发人员为了把他们久经考验的方法复制到面向对象架构中，会设计出许多复杂得难以想像的“聪明”方法。

**Poltergeist（恶作剧鬼）**：该反模式是指那些职责和有效生命周期都很有限的类。它们常常被用于启动其他对象的处理。重构方案中将这些责任重新分配给具有更长生命的对象来消除这些Poltergeist。

**Boat Anchor（船锚）**：它是指在当前项目中没有起到有益作用的那些软件或硬件。而且它往往具有相当高的成本，这使得当初对它的采购显得更为具有讽刺意味。

**Golden Hammer（金锤）**：它是指把一种熟悉的技术或概念强迫性地应用于许多软件问题。解决方案涉及通过教育、培训和读书研讨组来让开发人员了解其他的替代技术和方法，扩展他们的知识面。

**Dead End（死胡同）**：如果对一个可复用构件进行修改后它不再受供应商的支持，那么对它的修改就会形成Dead End。做出这些修改后，提供支持的负担就会转移到应用系统开发人员和维护人员身上。对可复用构件的改进将难以被集成，而且如果出现支持问题，就会被归咎到这些修改。 70

**Spaghetti Code（面条代码）**：即兴生成的软件结构导致难以扩展和优化代码。经常性的代码重构可以改善软件结构，为软件维护提供支持，并允许采用迭代式开发。

**Input Kludge（输入拼凑）**：未能通过直接行为测试的软件可能就是Input Kludge的例子。在采用即兴实现的算法来处理程序输入的时候就有可能发生这种反模式。

**Walking through a Minefield（穿越雷区）**：使用当今的软件技术就像是穿越高科技雷区[Beizer 1997a]。发布的软件产品中有无数的缺陷；实际上，专家估计在原始代码的每行中包含2~5个缺陷。

**Cut-and-Paste Programming（剪贴编程）**：通过剪贴语句形成的代码复用会导致显著的维护问题。包括黑盒复用在内的其他复用形式则可以通过使用相同的源代码、测试和文档来减少维护工作。

**Mushroom Management（蘑菇管理）**：在某些架构和管理圈子中，有明确的政策要把系统开发人员和最终用户隔离开。需求是通过一些中间人传递的二手信息，这些中间人包括架构师、项目经理或需求分析师。

除了上述这些反模式，本章还包括一些反映其他常见问题和解决方案的小型反模式。 71

反模式

## 5.3　The Blob（胖球）

**反模式名称**：The Blob
**别名**：Winnebago（温内贝戈人）[Akroyd 1996]和The God Class（上帝类）[Riel 1996]
**最常见规模**：应用层
**重构方案名称**：Refactoring of Responsibilities（职责重构）
**重构方案类型**：软件
**根源**：懒惰、匆忙
**不平衡的力量**：功能管理、性能管理、复杂性管理
**轶事证据**：“这个类确实就是我们架构的心脏。”

### 5.3.1　背景

还记得原版的黑白电影*The Blob*吗？也许你只看过最近重拍的。不管是哪一版本，故事情节几乎是一样的：一个水滴那么大的、果冻样子的外星生命从外太空来到了地球。每当这个果冻吃掉一些东西（通常是毫不戒备的地球人），它就会长大。这时，不肯相信他人的地球人陷入恐慌，没有人相信那个知道来龙去脉的疯狂科学家。许多人在恢复理智之前就被吃掉了。最后，怪物长大到可以把整个地球的生物都吃光。这部电影是The Blob反模式的一个很好的类比，这种反模式可以吞食掉整个面向对象架构（见图5-1）。

图5-1　The Blob（胖球）

## 5.3.2　一般形式

当一个类垄断了处理过程，而其他类主要只是用于封装数据的时候，在设计中就会发现The Blob。该反模式的特点是它的类图由单个复杂的控制器类和围绕着它的简单数据类构成，如图5-2所示。这里的关键问题在于责任的主体被分配到单个类中。 73

一般来说，虽然The Blob也许是用对象图示法来表示，以面向对象语言来实现的，但它仍然是一个过程式的设计。过程式设计把过程和数据隔离开，而面向对象设计融合了过程和数据模型并进行分割。The Blob包含处理过程的主体，而其他类则用于容纳数据。使用The Blob的架构把过程和数据隔离开，也就是说，它们的架构是过程式风格的而不是面向对象的。

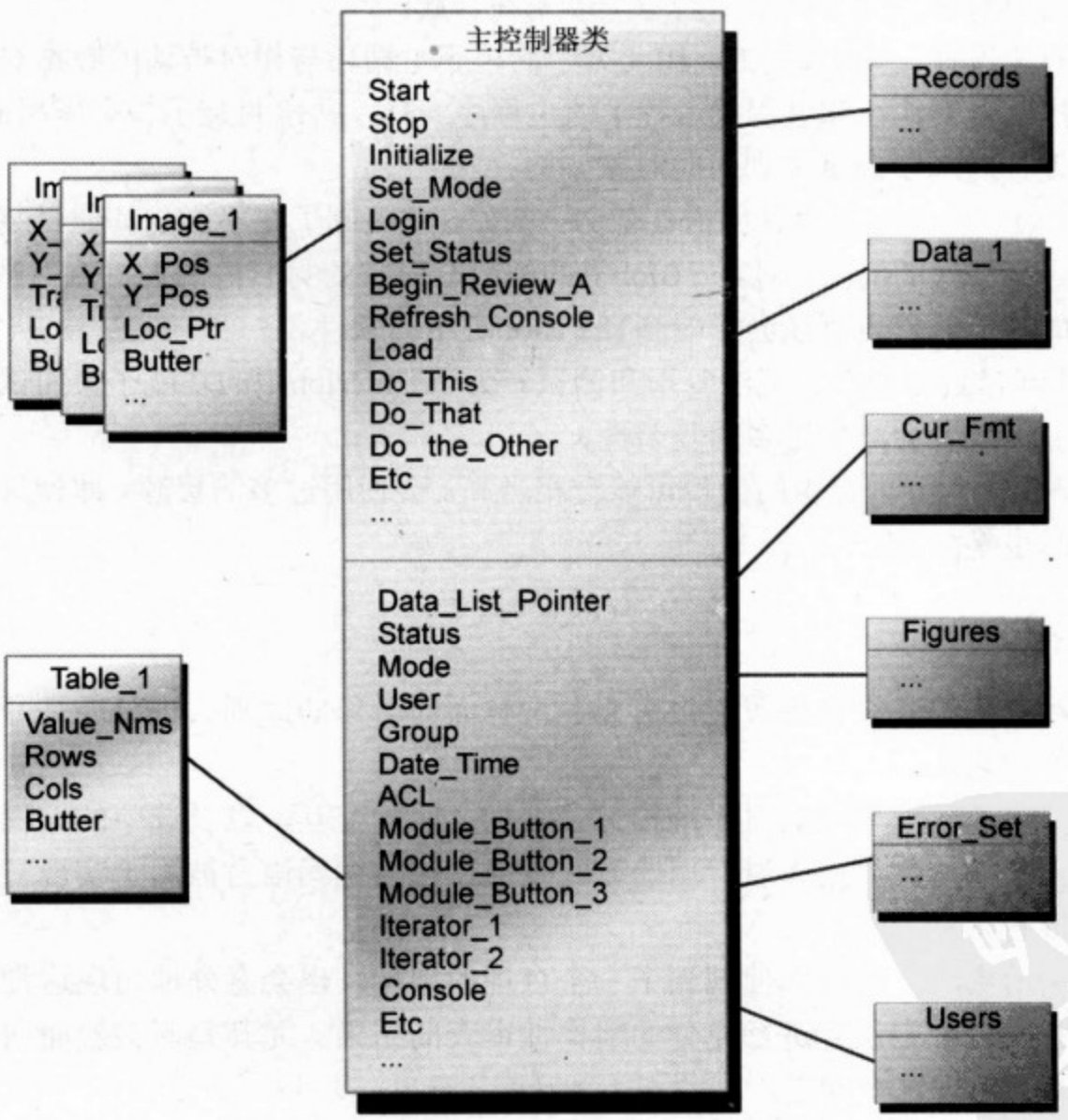



图5-2　控制器类

The Blob可能是需求分配不当的结果。例如，它也许是一个被赋予系统控制和系统管理职责的软件模块，而模块的这些职责与系统中许多其他部分的职责相互重叠。它也常常可能是从概念验证代码逐渐发展成原型并最终成为产品系统的迭代式开发过程的结果。主要以图形用户界面为中心的编程语言（如Visual Basic）常常会加剧这种情况。这样的语言允许在增量式开发或原型构 74

造过程中逐渐增加一个简单对话框的功能，也就是增加它的用途。在这样的系统演化过程中没有重新分割对职责的分配，于是某个模块成为了支配性的。The Blob往往伴随着不必要的代码，使得很难区分The Blob类的有用功能与不再使用的代码（参阅Lava Flow反模式）。

### 5.3.3　症状和后果

- 单个类拥有大量的属性或操作，或两者都有。具有60个以上属性和操作的类通常就表示The Blob的存在。
- 在单个类中封装了异类的、不相关的属性和操作集。属性和操作在总体上缺乏相关性是
75 The Blob的典型特征。
- 单个控制器类与多个简单的数据对象类联系在一起。
- 缺乏面向对象设计。系统就是The Blob类中的程序主循环与相对被动的数据对象联系在一起。单个控制器类往往和过程式编程中的主程序一样，几乎封装了整个应用的所有功能。
- 迁移遗留设计时未正确重构到面向对象架构。
- The Blob影响了面向对象设计的内在优势。例如，它限制了在不影响其他封装对象的条件下对系统进行修改的能力。对The Blob类的修改会对它的封装内的大量软件产生影响。修改系统中的其他对象也可能会影响到The Blob类中的软件。
- The Blob类通常过于复杂，无法复用和测试。为了The Blob中的功能子集而试图复用它往往是低效的，或者会产生过多的复杂性。
- 把The Blob类载入到内存中的代价可能会相当高，会使用过多的资源，即使只需要其中的简单操作时也是如此。

### 5.3.4　典型原因

- 缺乏面向对象架构。设计者可能没有充分理解面向对象的原则。也可能是开发团队缺乏适当的抽象技能。
- 缺乏（任何形式的）架构。对系统构件、它们之间的交互，以及选中的编程语言的特定用途缺乏定义。这使得程序以专用的方式发展，因为编程语言被用于实现它们的本来意图之外的目的。
- 缺乏对架构的实施。有时即使规划了一个合理的架构，也会意外地出现这种反模式。它可能是在开发过程中没有进行充分的架构性审查的结果。尤其是新接触面向对象的开发团队尤其容易出现这个问题。
- 干预过少。在迭代式项目中，开发人员倾向于给已有的可工作类一点一点增加功能，而不是增加新类或者为了更有效地分配职责而修改类层次关系。
- 需求说明灾难。有些时候，The Blob来源于说明需求的方式。如果系统需求规定了一个过
76 程式解决方案，在需求分析阶段可能就会在架构方面做出难以改变的承诺。把定义系统架构作为需求分析工作的一部分通常是不恰当的做法，往往会导致The Blob反模式，甚至可能会更糟。

### 5.3.5　已知例外

在包装遗留系统时，The Blob反模式可能是可以接受的。这时可以不需要软件分割，而只需要最终的一层代码来让遗留系统具有更高的可访问性。

### 5.3.6　重构方案

和本节的大多数反模式一样，解决方案中包含了一个重构形式。关键是把一些行为从The Blob中移出去。合适的做法也许是把一些行为重新分配到某些封装的数据对象，让它们起更多的作用，而The Blob类不再那么复杂。职责重构的方法如下所述：

（1）根据契约确定或分类相关的属性和操作。这些契约本身应该是相互关联的，直接与整体系统中某个共同的焦点、行为或功能相联系。例如，一个图书馆架构图可能会被一个名为LIBRARY的潜在The Blob类所代表。在图5-3所示的例子中，LIBRARY类封装了整个系统的所有功能。这样，重构的第一步就是确定代表了契约的相关操作和属性集合。在这个例子中，我们可以把与目录管理相关的操作，如Sort_Catalog和Search_Catalog集合到一起，如图5-4所示。我们还可以确定与单个条目相关的所有操作和属性，例如Print_Item、Delete_Item，等等。

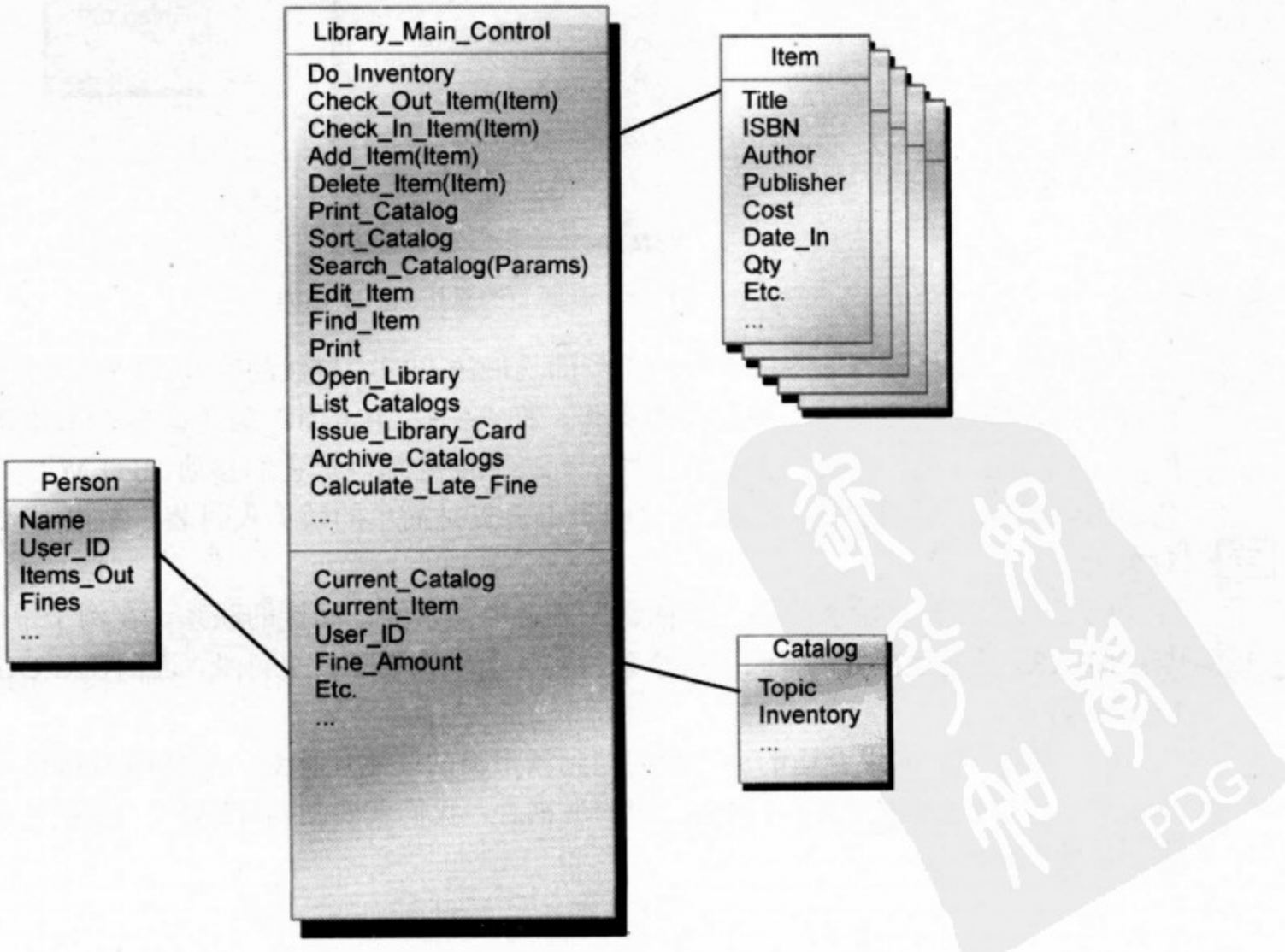



图5-3　The Blob类Library

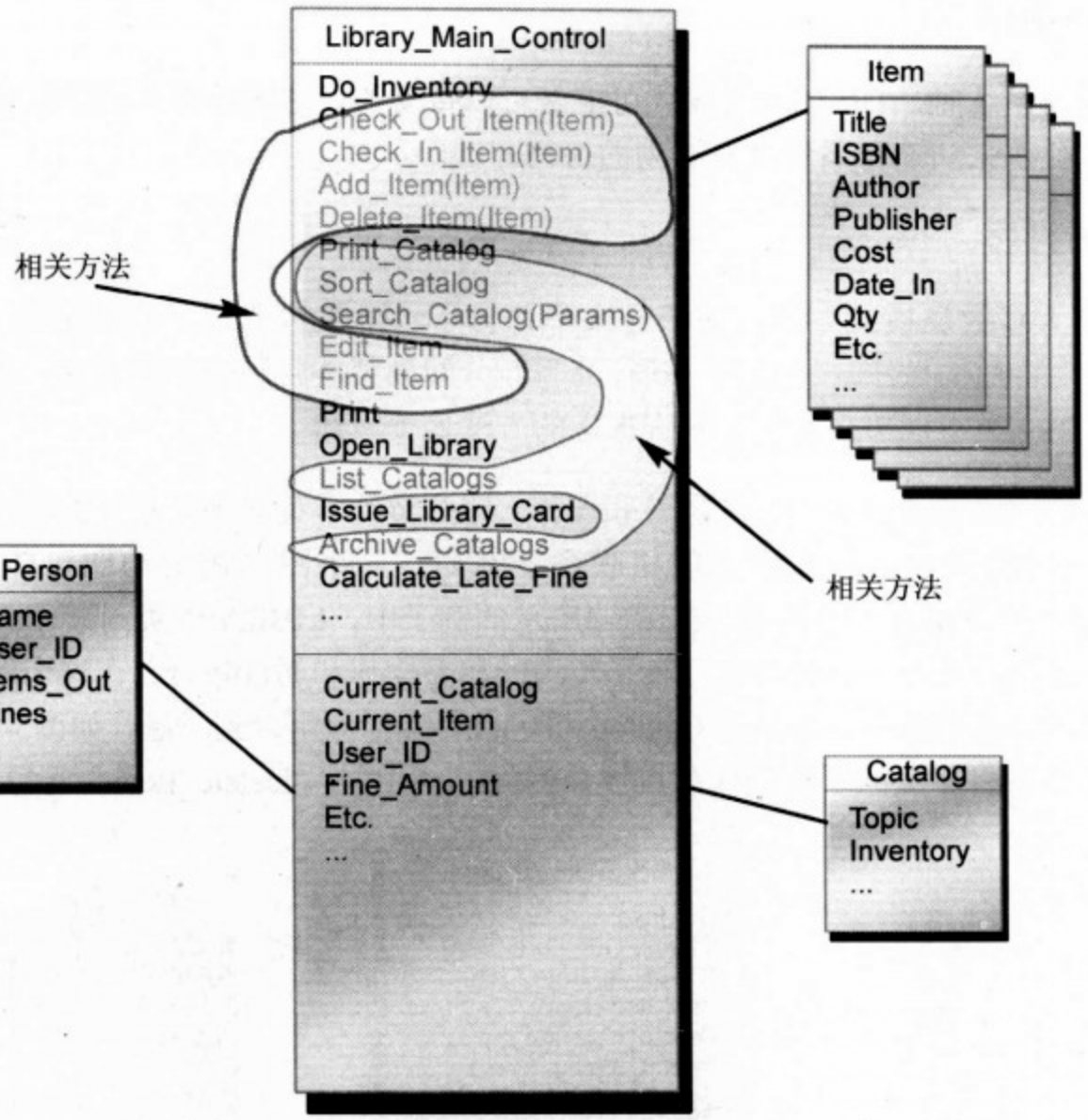



图5-4　根据契约重组The Blob类

（2）第二步是寻找这些根据契约得到的功能集合的“自然的家”，并把它们迁移过去。在这个例子中，我们收集了与目录相关的操作，把它们从LIBRARY类迁移到CATALOG类，如图5-5所示。我们对与条目有关的操作和属性进行类似的处理，把它们移动到ITEM类。这样既可以简化LIBRARY类，也让ITEM和CATALOG类不再只是简单的数据表封装。结果形成了更好的面向对象设计。

77

（3）第三步是移除所有的“远耦合”或者说冗余的、间接的联系。在例子中，ITEM类最初与LIBRARY类是远耦合的，因为每个条目实际上应该属于一个目录，而CATALOG类才直接属于LIBRARY类。

（4）接下来，如果合适的话，我们把到派生类的联系迁移到一个共同的基类。在例子中，一旦去除了在LIBRARY类和ITEM类之间的远耦合，我们就要把与ITEM类的关系迁移到CATALOG类，如图5-6所示。

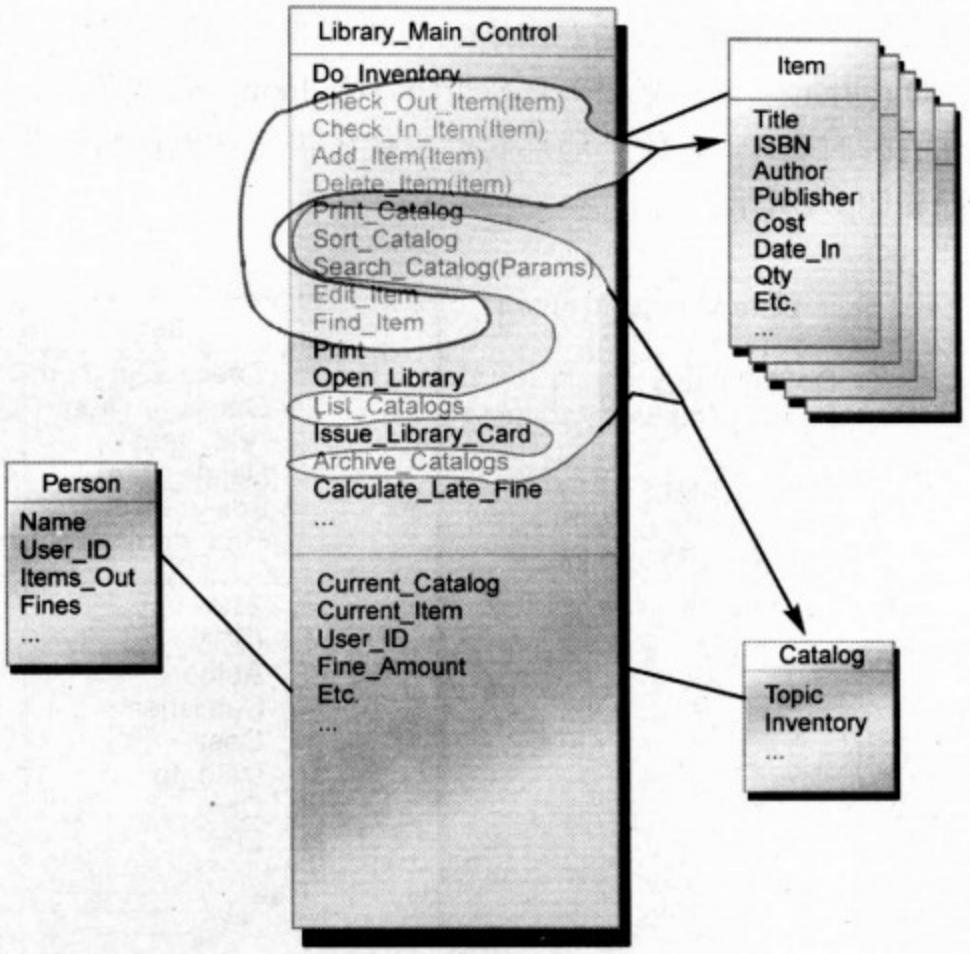



图5-5　迁移契约集合

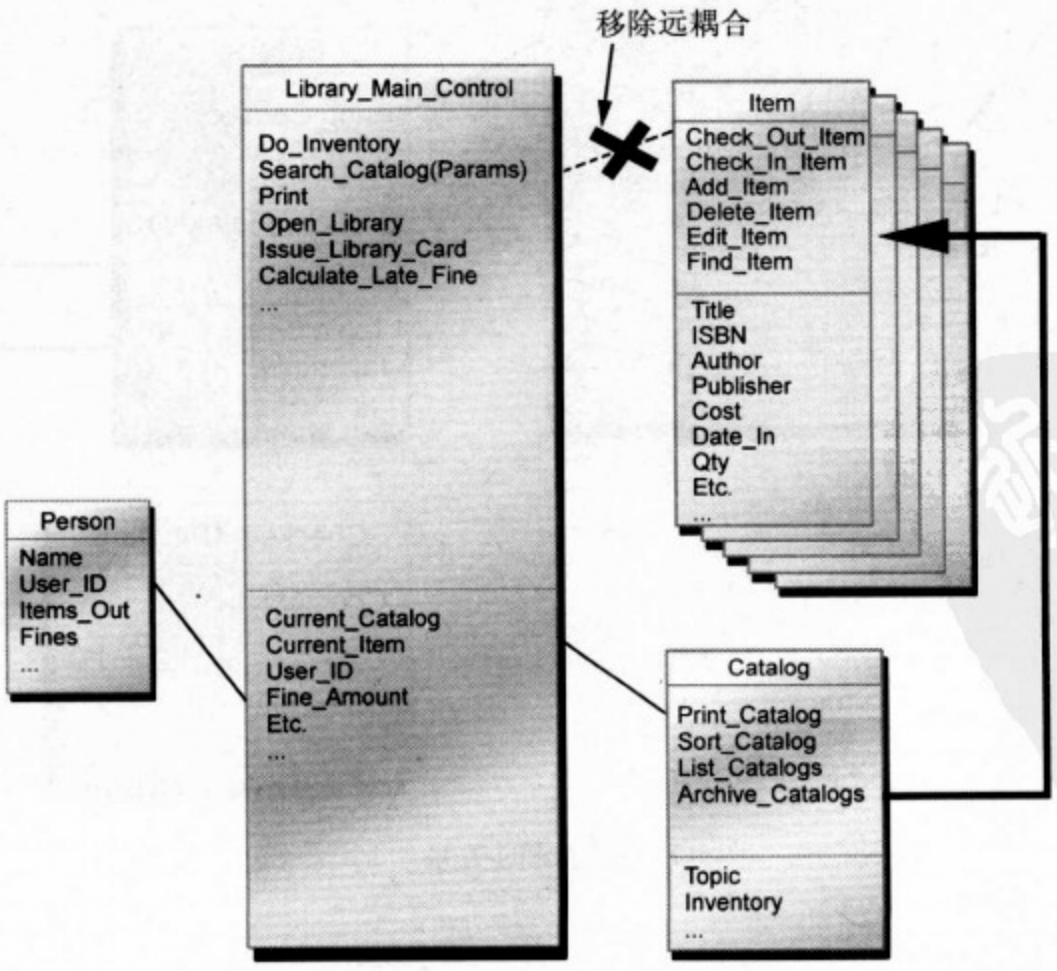



图5-6　移除远耦合

（5）最后，我们移除所有的瞬时关系，用适当的属性类型说明和操作参数来替代它们。在我们的例子中，Check_Out_Item（借出条目）和Search_For_Item（查找条目）是瞬时处理过程，可以被移动到一个独立的瞬时类中，它使用局部属性来代表借出实例的特定位置或查找实例的特定查找条件。这一过程如图5-7所示。

78

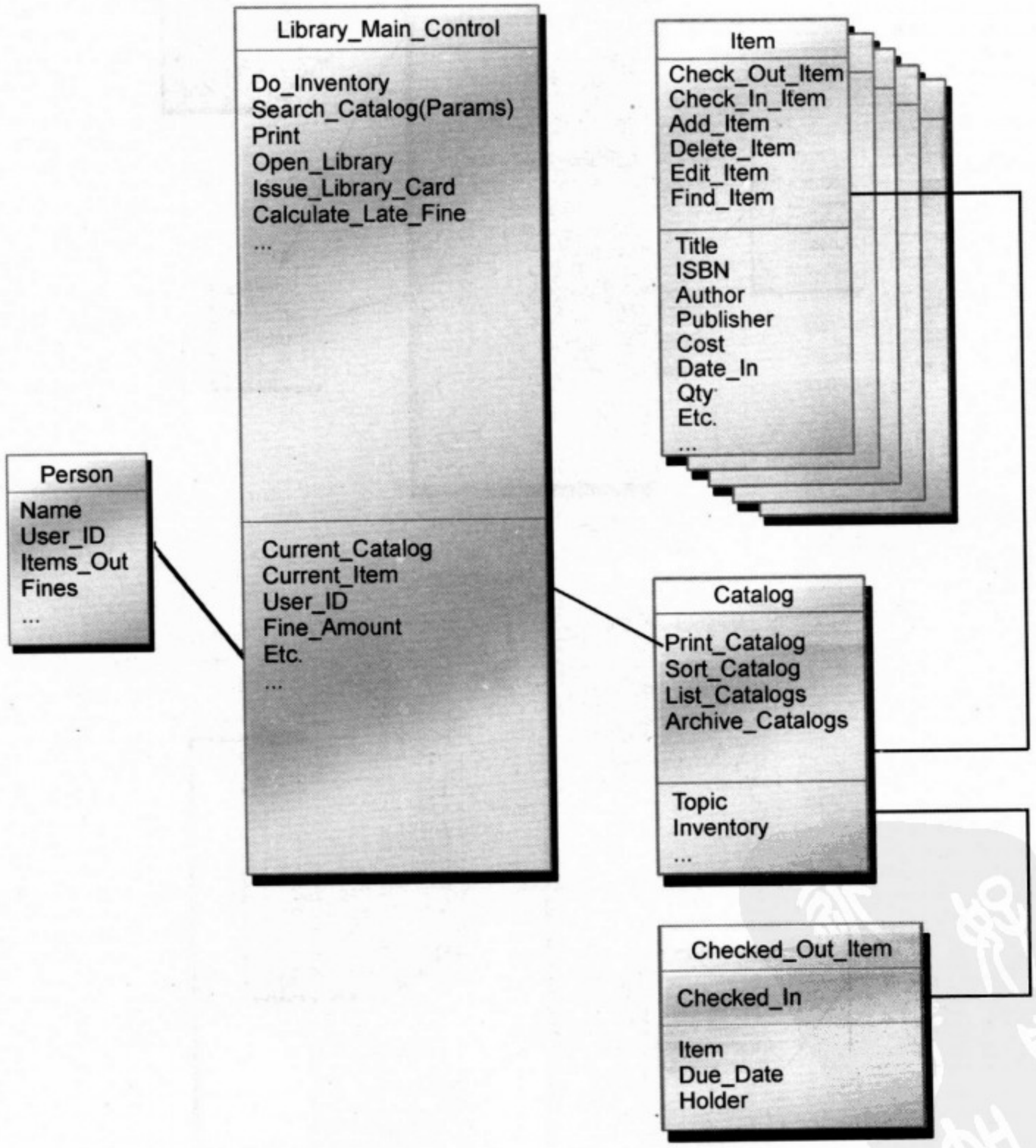



图5-7　移除瞬时关系

## 5.3.7　变化

有时，对一个由The Blob类和支持数据类构成的系统，需要做太多的工作才能完成对类架构

的重构。有可能采用一种提供一个“80%”方案的替代方法。可以减少The Blob类的功能，让它从控制器类变成协调器类，而不是完全重构整个类层次关系。让最初的The Blob类负责管理系统的功能，对数据类则扩充一些处理过程。数据类在修改后的协调器类的指导下进行操作。这个过程可以保持原始的类层次关系，而只是把处理功能从The Blob类移动到某些数据封装类中。 79

Riel发现了The Blob反模式的两种主要形式。他把它们分别称为行为形式和数据形式的God Class [Riel 1996]。行为形式是包含了集中式处理过程的对象，它与系统的大多数其他部分进行交互。而数据形式的对象则是包含的数据被系统的大部分其他对象所使用。Riel提出了一些用于检测和重构God Class的面向对象探索方法。

### 5.3.8　对其他视角和规模的适用性

要在一开始就防止The Blob反模式的出现，架构视角和管理视角都扮演着关键的角色。避免 80
The Blob的出现可能要求持续地对架构进行维持，以保证对职责进行充分的分散。通过架构视角才能发现正在出现的The Blob。通过成熟的面向对象分析和设计过程，以及警觉的、充分理解了设计的管理者，开发人员可以防止培养出The Blob类。

在大多数情况下最重要的因素是，建立适当设计的成本比在实现后对设计进行返工的成本要低廉得多。先期投资于良好架构和对团队进行教育，可以保证项目避免The Blob和大多数其他反模式。随便问一个保险推销员，他都会告诉你大部分人都是在需要获得保险赔偿之后才购买的，那时他们已经更穷，不过也更明智了。 81 ~ 82

### 5.3.9　示例

一个为处理模块提供界面的GUI模块逐渐取代了后台处理模块的处理功能。一个这种情况的示例是用于客户数据输入/获取的PowerBuilder界面。这个界面可以：

（1）显示数据。
（2）编辑数据。
（3）进行简单的类型验证。开发人员然后增加了相当于决策引擎的功能：
- 复杂验证。
- 使用验证过的数据来评估后续操作的算法。

（4）开发人员又获得了新的需求：
- 把该GUI扩展到3个表单。
- 让它成为脚本驱动的（包括开发一个脚本引擎）。
- 给决策引擎增加新算法。

开发人员扩展了当前模块来加入所有这些功能。结果开发出一个模块而不是开发多个模块，

如图5-8所示。如果对该应用进行了架构和设计，则更容易进行维护和扩展。它看起来会与图5-9
83 相似。

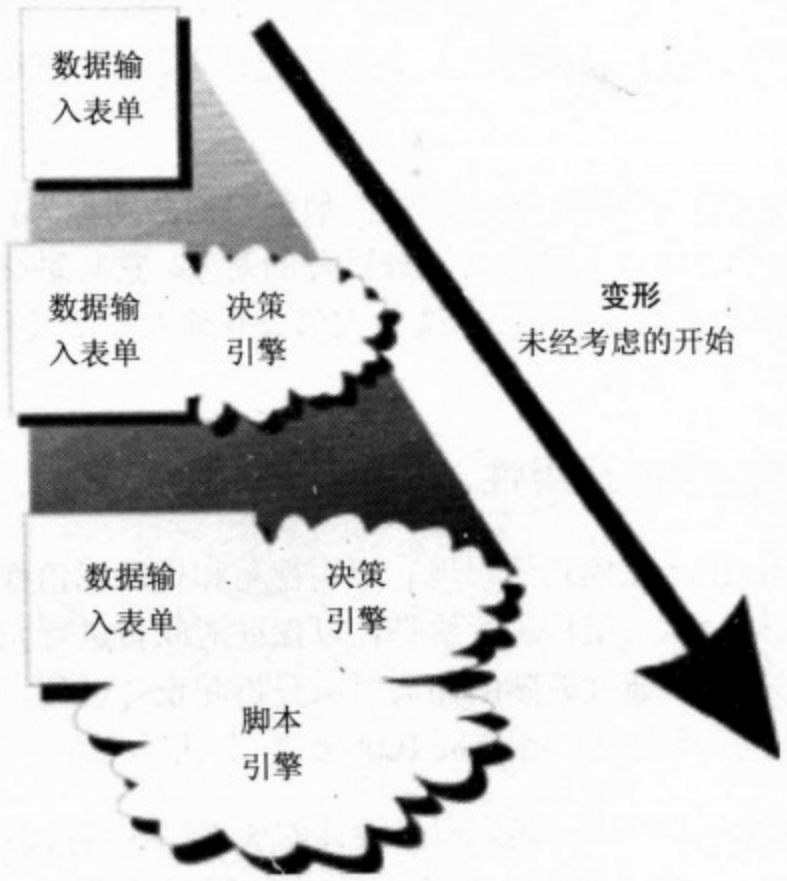



图5-8　GUI原型的生命历程


84

图5-9　应用原型的生命历程

### 小型反模式：Continuous Obsolescence（持续过时）

**反模式问题**

技术变化非常迅速，以至于开发人员往往难以跟上软件的当前版本，无法找到可以共同工作的产品发布的组合。考虑到每个商业产品线都要通过新版本来进行发展，开发人员要面对的情况就越发艰难。找到可以成功互操作的兼容产品发布则更加困难。

Java是这一现象的著名例子，它每过几个月就会出现一个新版本。例如，在一本有关Java 1.x的书出版时，新的JDK已经让这些信息过时了。Java并不是惟一有此问题的，还有很多其他技术也有Continuous Obsolescence问题。最公然这样做的例子是那些在商标名称中嵌入发布年号的产品，例如Product98。采用这种方式，这些产品几乎是在炫耀自己的过时。另一个例子就是Microsoft动态技术的发展过程：

DDE
OLE 1.0
OLE 2.0
COM
ActiveX
DCOM
COM+

从技术销售商的角度来看，有两个关键的因素：思想共享和市场份额。快速的革新要求客户专心关注才能跟上最新的产品特性、声明和术语。对那些遵循某种技术的人来说，快速革新对思想的共享是有贡献的；也就是说，总有关于某种技术的新消息。一旦获得了占据统治地位的市场份额，供应商的收入就主要来自于让早期产品发布过时和对它进行替换。技术过时（或被认为过时）越迅速，收入就越高。

85

**重构方案**

开放式系统标准是技术市场的重要稳定因素。联盟标准是行业中花费大量时间和投资之后达成一致的产物。随着某项技术成为主流，联合的市场推广活动会让用户了解和接受该技术。这一过程的内在惯性有利于消费者，因为一旦某个供应商的产品符合标准，制造者就不太可能改变该产品符合标准的特性。

快速过时的技术所带来的优点是过渡性的。架构师和开发人员应该依赖稳定的接口或它们所控制的内容。开放式系统标准提供了对技术市场稳定性的度量。如果缺乏它，技术市场就会混乱无序。

**变化**

Wolf Ticket小型反模式（第6章）说明了一些方法，消费者可以使用它们来影响产品的发展方向，获得更高的产品质量。

86

反模式

## 5.4　Lava Flow（岩浆流）

**反模式名称**：Lava Flow
**别名**：Dead Code（死代码）
**最常见规模**：应用层
**重构方案名称**：Architectural Configuration Management（架构配置管理）
**重构方案类型**：过程
**根源**：贪婪、懒惰
**不平衡的力量**：功能管理、性能管理、复杂性管理
**轶事证据**："哦，那些啊。Ray和Emil（他们已经不在这个公司了）写了那段例程，当时Jim（他上个月离开了）正在尝试绕过Irene（她现在也去了另一个部门）的输入处理代码。我想现在没什么地方还使用它，不过我也不确定。Irene并没有很清楚地给它写文档，所以我们估计现在还是应该让它保持原样。毕竟这东西可以工作，不是吗？"

### 5.4.1　背景

在一次数据挖掘的探索中，我们开始寻找为特定类型的系统开发标准界面所需的深入理解。我们挖掘的系统与我们希望最终可以支持正在研究的那些标准的系统非常相似。它也是一个面向研究的系统，而且非常复杂。随着研究的深入，我们与许多开发人员进行了交谈，他们分别和我们打印出的无数页代码中的特定部分相关。我们一再得到相同的回答："我不知道那个类的目的，
87 我来之前它就写好了。"我们渐渐认识到，这个复杂系统中有大约30%~50%的实际代码没有被当前处理该系统的任何人所理解或写下文档。而且随着我们的分析，我们发现那些可疑的代码实际上在当前系统中没有任何作用；它们只是早已离开的开发人员遗留下来的以前采用过的尝试或方法。当前的开发人员虽然很聪明，但是也不愿意修改或删除不是他们编写的代码，或者不了解其用途的代码。因为他们担心会破坏某些地方而不知道其原因或者不知道该如何修复它。

这时，我们开始称这些代码块为代码"岩浆"，用来说明它刚出现时具有可变的本质，而一旦固化了就像玄武岩一样坚硬而难以去除。我们突然认识到发现了一个潜在的反模式。

在之后大约一年的时间里，我们又进行过几次数据挖掘探索和界面设计工作。我们非常频繁地遇到同样的这个模式，以至于我们整个部门都把它称为Lava Flow（见图5-10）。

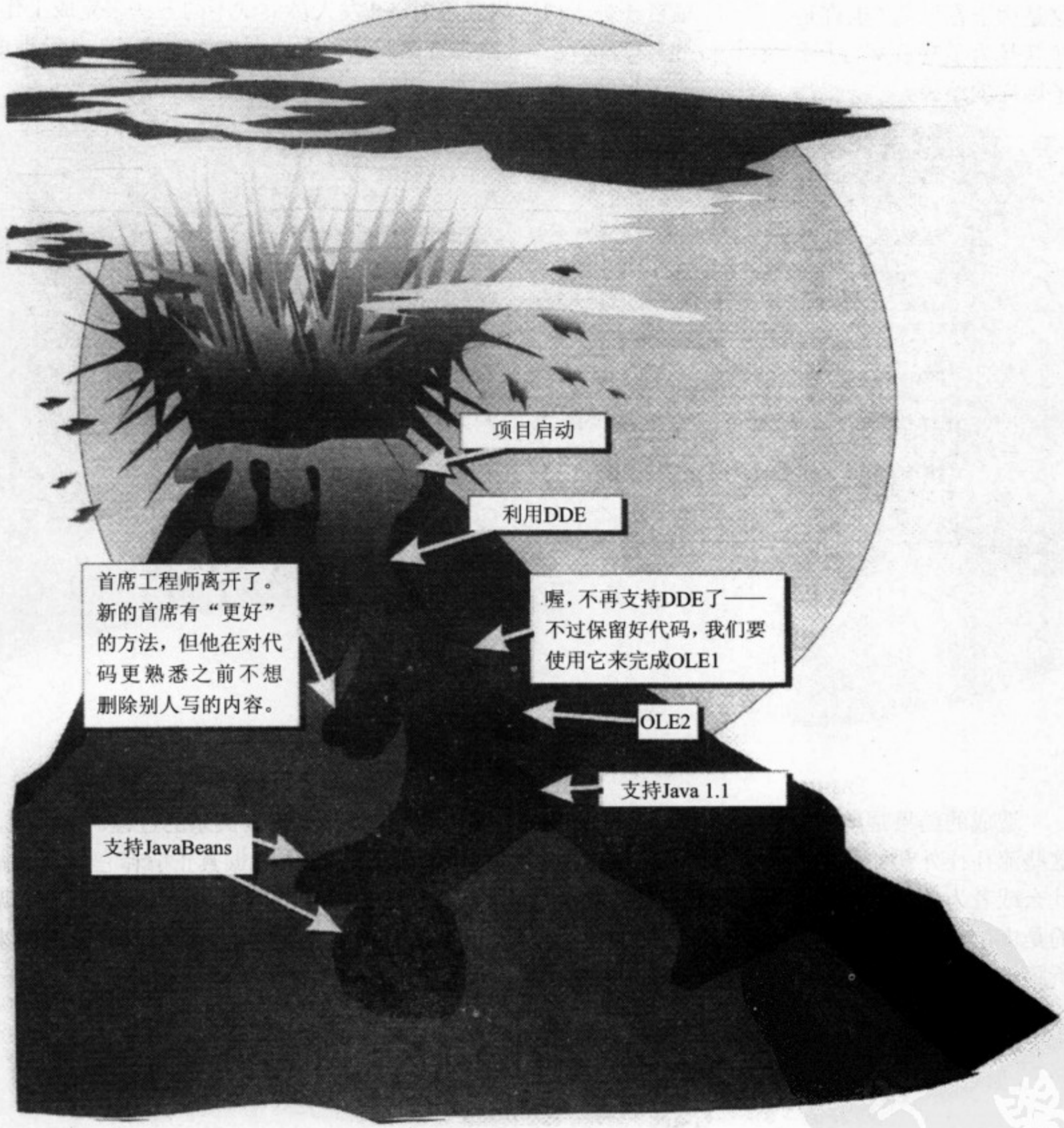



图5-10　过时的技术和遗忘的扩展形成的Lava Flow在它的尾迹中留下固化的死代码

## 5.4.2　一般形式

在开始于研究活动最后以产品结束的系统中，常常可以发现Lava Flow反模式。它的特点是在代码中散布着过去的开发版本形成的类似岩浆的代码“流”，而现在它们固化成像玄武岩一样，不可移动、逐渐失去作用。没有人对它们还有多少印象，或者根本就没有印象了（见图5-11）。

这是由于在以前（也许是侏罗纪）项目还处于研究模式下时，开发人员尝试不同方法来完成工作，尤其是为了赶着交付某种演示时，把合理的设计实践都置诸脑后，文档也被牺牲掉了，从而造成了这样的结果。

```
...
// This class was written by someone earlier (Alex?) to manage the indexing
// or something (maybe). It's probably important. Don't delete. I don't
// think it's used anywhere - at least not in the new MacroINdexer module which
// may actually replace whatever this was used for...
class IndexFrame extends Frame
{
  // IndexFrame constructor
  //-------------------------------------------------------------------
  public IndexFrame(String index_parameter_1)
  {
  // Note: need to add additional stuff here...
  super (str);
  }
  //-------------------------------------------------------------------
...
```

图5-11　Lava Flow源代码列表

造成的结果是严重破碎的代码、任意改变的类和与系统整体没有清晰关系的过程。实际上，这些流往往外观如同乱麻一般非常复杂，让人以为它们很重要，但没人能够真正解释它们到底做什么或者为什么会存在。有时，某个上了年纪的头发花白的开发隐士会想起某些细节，但更常见的是由于有疑问的代码“并没有真的造成什么破坏，可能实际上是很关键的，我们只是没时间来折腾它”，所有人都决定“让它保持原样”。

“架构是关于如何浪费空间的艺术。”

——Phillip Johnson（美国著名建筑师）

88 ~ 89

虽然解剖这些岩浆流，研究它们的人类学因素会很有趣，但在进度表上一般不会有足够的时间来绕这样的路。开发人员通常会采取权宜之计，巧妙地绕过它们。

但是，该反模式在概念验证或原型代码迅速发展成产品的创新设计工场中令人难以置信地常见。它是一个不好的设计，以下是几个关键原因：

- 对Lava Flow进行分析、检验和测试的成本很高昂。所有这些工作都是没有价值的，完全

就是浪费。实际上，几乎不可能进行检验和测试。

- 读取Lava Flow代码到内存中的代价也很高，会浪费重要的资源，影响性能。
- 和许多反模式一样，你会失去面向对象设计的内在优势。在这种情况下，你失去了在不扩散Lava Flow块的情况下进行模块化和复用的能力。

90

### 5.4.3　症状和后果

- 系统中频繁出现无法解释其合理性的变量和代码碎片。
- 没有文档的、似乎重要的复杂函数、类或代码段，它们与系统架构之间没有清晰的关系。
- 非常松散的、“进化中的”系统架构。
- 整段注释掉代码而没有解释或文档。
- 大量“正在变化”或“待替换”的代码区。
- 未使用的（死）代码，简单地留在那里。
- 头文件中存在未使用的、无法说明的或过时的接口。
- 如果移除了现有的Lava Flow代码，由于代码在其他地方被复用，它仍然可能会扩散。
- 如果没有对导致Lava Flow的过程进行检查，而后续开发人员在分析原始流时太匆忙或太不情愿，结果在试图绕过原始流时继续产生新的流，可能会导致产生指数形式的增长。这会让问题更复杂。
- 随着流的复杂化和硬化，很快就会无法为代码编写文档，也无法充分理解它的架构以便加以改进。

### 5.4.4　典型原因

- 在未考虑配置管理的情况下把研发代码放入到产品中。
- 对未完成的代码进行未受控制的分发。为实现某种功能而实现多种试验方法。
- 单个开发人员（也称为独狼，lone wolf）编写代码。
- 缺乏配置管理或没有充分遵循过程管理策略。
- 缺乏架构或者采用非架构驱动的开发。这一点对临时性很高的开发团队来说尤其普遍。
- 反复性的开发过程。软件项目的目标常常是不清楚的，或者反复发生变化。要解决变化，项目必须返工、倒退和开发原型。为了应对演示的最终期限，常见的做法是用很少的时间地对代码做出匆忙的改变来解决眼前的问题。永远都不会清理代码，让对架构的考虑和文档化工作被无限期地推迟。

91

- 架构疤痕。有时，会在进行了一些开发后发现在需求分析时做出的架构承诺是不可行的。系统架构可能需要重新配置，但很少会去掉那些内联的错误。注释掉不需要的代码甚至都有可能是不可行的，尤其是在现代开发环境中，可能有数百个文件包含系统的代码。“谁会看所有这些文件？把它们链接进来就是了！”

### 5.4.5　已知例外

研发环境中的小规模、用后抛弃的原型非常适合使用Lava Flow反模式。在这种环境中，迅速开发是很重要的，其结果并不要求是可维持的。

### 5.4.6　重构方案

只有一种可靠的方法来防止Lava Flow反模式：保证在产品代码开发前建立健全的架构。该架构必须受到配置管理过程的支撑，这个管理过程要保证开发符合架构规定，而且可以容纳“任务蔓延”（变化的需求）。如果一开始进行架构考虑时采取欺骗的态度，最终会开发不属于目标架构的代码，从而产生冗余的代码或死代码。经过一段时间，死代码就会成为分析、测试和修改时的问题。

在出现了Lava Flow的地方，治疗过程会非常痛苦。重要的原则是在活跃的开发中避免架构变化。这一原则尤其适用于计算体系，也就是定义系统集成方案的软件接口。项目管理过程必须把开发推迟到定义了清晰的架构并让开发人员都了解它之后。定义架构可能需要进行数个系统发现活动。要通过系统发现来找到那些实际使用的、对系统而言不可缺少的构件。系统发现还会确定那些可以被安全删除的代码行。这项活动相当冗长，它可能会需要一个有经验的软件侦探的调查技能。

92

在消除可疑的死代码时，有可能会产生错误。发生这种事的时候，要避免在充分了解出错原因之前马上修正出现的症状的冲动。要研究问题的依赖关系，这可以帮助你更好地定义目标架构。

要避免Lava Flow，重要的是建立系统层次的软件接口。它应该是稳定的、定义良好的，而且具有清楚的文档。从长远的角度看，与费力地排除坚硬的Lava Flow代码块相比，在软件接口质量上进行先期投入的成本可能要低很多倍。

类似于源代码控制系统的根据可以帮助进行配置管理。大多数Unix环境都捆绑了源代码管理系统，提供了基本的能力来记录受配置控制的文件的更新历史。

### 5.4.7　示例

我们最近参与了一次数据挖掘探索，试图从最初的接口架构中确定出进化后的接口，该架构是我们原创的，正处于不断更新的过程中。对该系统进行挖掘的原因是因为开发人员使用我们的原始架构的独特方式让我们很着迷：本质上，他们用我们的通用应用间架构建立了一个准事件服务。

在研究他们的系统时，我们遇到了大段代码的阻碍。这些代码段似乎对我们期望发现的整体架构没有贡献。它们在某种程度上缺乏内聚性，而且只有很少的文档或者根本没有。当我们问当前的开发人员关于其中一些部分的问题时，得到的回答是：“那些吗？嗯，我们不再使用那个方

法了。Reggie试过某个方法，但是我们提出了更好的办法。不过我想Reggie的其他代码可能会依赖那些内容，所以我们没有删除任何内容。”当我们更深入地研究时，才发现Reggie甚至已经不再参与开发，有一段时间没有来过了，于是那些代码段已经放在那里几个月了。

经过两天的代码检查，我们认识到组成系统的大部分代码与我们研究过的代码非常相似：本质上完全是Lava Flow。我们收集到的有关他们到底是如何构造架构的信息非常少；因此，几乎 93
不可能进行挖掘。这时，我们基本上放弃了尝试挖掘代码，转而关注当前开发人员对“到底”做了什么的解释，希望能够以某种方式把他们的工作编制成接口扩展，让我们可以把它结合到接下来对我们的通用应用间框架的修改中。

我们的解决方法是分离出对他们开发的系统具有最深入理解的单个关键人物，然后和他共同编写IDL。表面上看，我们共同编写的IDL的目的是支持数周后的一次决定性演示。利用Fire Drill小型反模式，我们让系统开发人员为这次演示给他们的产品迅速建立了一个CORBA封装，从而验证了我们的IDL。很多人都熬了不少的夜，但是演示很成功。当然这个解决方法还有另一个作用：我们最终结束于用IDL表示的接口，而它正是我们一开始就想发现的。

### 5.4.8　相关解决方案

在当前的竞争性世界上，尽量减少研发和产品之间的时间差常常是大家都想达到的目标。在很多行业中，它对公司的存亡至关重要。在处于这种情况下时，有时会在定制的配置管理过程中发现避免Lava Flow的组合措施，这种配置管理过程在建立原型的阶段就设置了一些限制性控制，类似于把它“挂钩”到真实的、产品级的开发中，但没有完全限制研发的试验性本质。在可能的情况下，自动化可以扮演很重要的角色，但是关键之处在于定制一个一旦系统移动到产品环境中，就可以很容易扩展成完整配置管理系统的准配置管理过程。要解决的问题就是在两种成本之间取得平衡，一种是配置管理对创造性过程的阻碍所造成的成本，另一种则是在创造性过程产生了一些有用的、可销售的内容后迅速获得对开发的配置管理控制的成本。

可以通过周期性地把原型系统映射到更新的系统架构中来促进这种方法，系统中应该包括有限的、但是标准化的内联代码文档。

### 5.4.9　对其他视角和规模的适用性

架构视角在从最开始就防止Lava Flow中扮演了关键的角色。管理者也可以承担在早期发现 94
Lava Flow或发现会导致Lava Flow的环境的职责。这些管理者还必须具有权威，可以在一发现Lava Flow时就踩下刹车，推迟其后的开发直到定义和推广了一个清楚的架构。

与大多数反模式一样，预防的成本总是低于纠正。所以先期投资于良好的架构和团队教育通常可以保证项目避免Lava Flow和大多数其他反模式。虽然这种起始成本不会马上显示出回报，但它的确是一种好投资。

## 小型反模式：Ambiguous Viewpoint（模糊视角）

**反模式问题**

面向对象分析和设计模型常常没有澄清该模型所表达的视角。默认情况下，OOAD模型采用了实现视角，而这个视角在很多时候很可能是最没有用的。混合的视角使得在接口和实现细节之间无法进行基本的隔离，而这种隔离本应是面向对象范型带来的首要益处。

**重构方案**

面向对象分析和设计有三个主要视角：业务视角、规范视角和实现视角。业务视角定义了信息和处理的用户模型。这是领域专家要维护和解释的模型（通常称为分析模型）。分析模型是信息系统最稳定的一些模型，是值得维持的。

如果模型不聚焦于所要求的观察角度，就没那么有用了。观察角度对信息进行了过滤。例如，对电话交换系统的类模型定义会根据下列观察角度所提供的焦点而不同：

- 电话用户，他关心打电话和获取条目化通话账单时的便利性。
- 电话接线员，他关心把用户连接到需要的号码。
- 电话记账部门，他们关心记账规则和记录用户打出的所有电话。

95

从上面三个观察角度出发会得到一些相同的类，但是不会很多。即使是那些相同的类，它们的方法也不完全相同。

规范视角关注软件接口。由于对象（作为抽象数据类型）的目的是把实现细节隐藏到接口之后，规范视角定义了对象系统中暴露出的抽象和行为。规范视角定义了系统中对象之间的软件边界。

实现视角定义了对象的内部细节。实现模型在实践中常被称为设计模型。必须随着对软件的开发和修改对设计模型不停地进行维护，才能让它成为软件的准确模型。由于过时的模型是没有用的，所以只有选定的设计模型，尤其是那些说明系统的某些复杂方面的设计模型才需要得到维护。

96

反模式

## 5.5 Functional Decomposition（功能分解）

**反模式名称**：Functional Decomposition

**别名**：No Object-Oriented AntiPattern（非面向对象反模式）、“No OO”（非面向对象）[Akroyd 1996]

**最常见规模**：应用层

**重构方案名称**：Object-Oriented Reengineering（面向对象再设计）

**重构方案类型**：过程

**根源**：贪婪、懒惰

**不平衡的力量**：复杂性管理、变化管理

**轶事证据**：“这是我们的‘主’例程，在这个名为LISTENER（监听器）的类中。”

### 5.5.1 背景

Functional Decomposition在过程式编程环境中是不错的做法。它对理解更大规模应用的模块化本质也是有益的。不过，它不能直接转换成类层次关系，而这就是问题的来源。在定义这个反模式时，作者们开始于Michael Akroyd对此主题的最初考虑。我们重新设定了它的格式以符合我们的模板，并用解说和图示进行了扩展。

### 5.5.2 一般形式

这个反模式是有经验的非面向对象开发人员使用面向对象语言来设计、实现一个应用时出现的问题。当开发人员习惯于使用一个“主”例程来调用许多个子例程时，他们往往会把每个子例程变成一个类，完全忽略掉类层次关系（也几乎全完全忽略掉面向对象）。结果代码在类结构上非常类似Pascal或FORTRAN这样的结构化语言。它会难以想像的复杂，因为那些机灵的过程式开发人员会想出非常聪明的办法来使用面向对象架构复制他们那些久经考验的（非面向对象）方法。

97

如果开发组以前使用C语言刚刚转换到C++，或者试图使用CORBA接口，或者刚实现某种被认为可以帮助他们的对象工具，就最可能遇到这种反模式。从长远角度来看，花钱进行面向对象培训或雇用采用对象进行思考的新程序员，可能更廉价一些。

### 5.5.3 症状和后果

- 采用“函数”名，例如Calculate_Interest或Display_Table作为名称的类可能就意味着存在这种反模式。
- 类的所有属性都是私有的，只在类的内部被使用。

- 只有单个操作，如一个函数的类。
- 令人难以置信的退化架构，完全丢弃了面向对象架构的要点。
- 完全没有利用面向对象原则如继承和多态。它的维护成本可能极其昂贵（首先如果它能工作的话；不过永远不要低估了逐渐跟不上技术发展的老程序员的灵活性）。
- 无法清晰地记录（甚至解释）系统是如何工作的。类模型完全没有意义。
- 根本不能期望获得软件复用。
- 测试人员感到挫折和无望。

### 5.5.4　典型原因

- *缺乏对面向对象的理解。*实现者没有了解面向对象的本质。这在开发人员从非面向对象编程语言切换到面向对象编程语言时相当常见。由于在架构、设计和实现范型上都要发生变化，一个公司可能要花3年时间才能完全达到面向对象。
- *缺乏对架构的实施。*当实现者不理解面向对象时，架构设计得有多好并不重要。实现者根本就不能理解他们到底在做什么。如果没有适当的监督，他们通常会找到某种办法来用他们了解的方法蒙混过关。

98

- *说明灾难。*有时，负责生成说明和需求的人并不一定有开发面向对象系统的经验。如果他们说明的系统在需求分析之前做出架构性承诺，就有可能而且常常会导致类似Functional Decomposition的反模式。

### 5.5.5　已知例外

在不要求面向对象的解决方案时，Functional Decomposition反模式没什么问题。这一例外可以被扩展到用于处理具有纯功能本质而被包装起来，以便为实现代码提供面向对象接口的实现方案。

### 5.5.6　重构方案

如果还有可能探知软件基本需求的内容，就为软件定义一个分析模型来从用户的角度解释软件的关键特性。这对发现特定代码集中很多软件构造的内在动机是很重要的，而这些动机随着时间的流逝可能已经被遗忘了。对Functional Decomposition反模式解决方案中的所有步骤，都要提供所使用的过程的详细文档以作为将来维护工作的基础。

接下来，建立一个结合了已有系统的主要部分的设计模型。不要太强调改善模型而是建立一个基础来解释系统中尽可能多的部分。理想情况下，设计模型可以证明大部分软件模块的正确性，或者至少说明它们的合理性。为已有的代码集建立设计模型是具有启发意义的；它提供了对整个系统的组织结构的深度理解。完全有理由预测认为系统中某些部分的存在原因已经无人知晓，也无法做出合理的推断。

对没能进入设计模型的类，应用下面这些处理原则：

（1）如果该类只有一个方法，就尝试把它作为某个已有类的组成部分。某些类被设计为其他 99
类的辅助类，把它们和它们辅助的基类合并起来往往更好一些。

（2）试着把数个类合并成一个满足某个设计目标的新类。这样做的目标是把几种类型的功能合并到单个类中。与先前较细粒度的那些类相比，它应该捕获了更广阔领域中的概念。例如，不要分别使用几个类来分别负责管理设备访问、过滤与设备交换的信息和控制设备，而是把它们合并到单个设备控制器对象中，该对象的方法可以完成之前被分散到多个类中的活动。

（3）如果类中没有任何形式的状态信息，就要考虑把它改写成函数。系统中潜在地有某些部分可能最好是被建模成函数，从而可以在系统的不同部位不受限制地访问它们。

对设计进行检查，找到相似的子系统。它们是复用的候选对象。作为程序维护的一部分，应该重构代码集以便在相似子系统之间复用代码（参阅Spaghetti Code反模式的解决方案以获得对软件重构的详细说明）。

### 5.5.7　示例

Functional Decomposition建立于用于操作数据的离散函数，例如可以通过使用Jackson结构化编程方法来建立。函数往往就是面向对象环境中的方法。两种环境中对函数的划分是根据不同范型进行的，从而产生了对函数及相关数据的不同分组。

图5-12中的简单例子显示了客户贷款场景的功能化版本：

（1）添加一个新客户。
（2）更新客户地址。
（3）计算给客户的一笔贷款。
（4）计算一笔贷款的利息。
（5）计算一笔贷款的还贷进度表。
（6）改变一个还贷进度表。

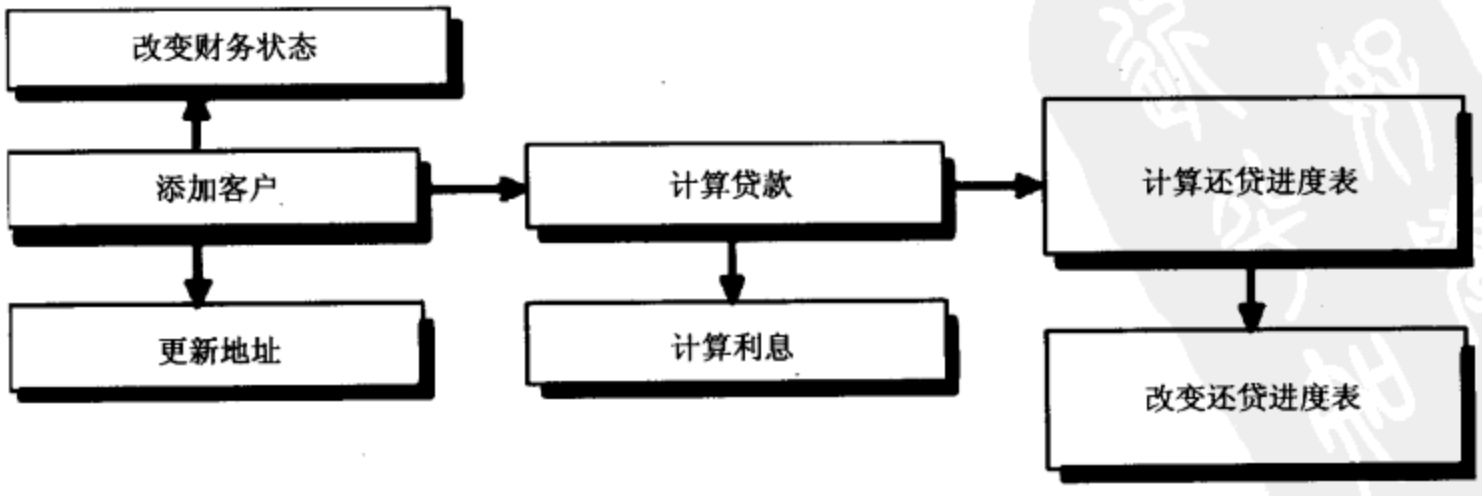



图5-12　客户贷款系统的功能分解

100 图5-13显示了客户贷款应用的面向对象视图。前一个图中的功能被映射成了对象方法。

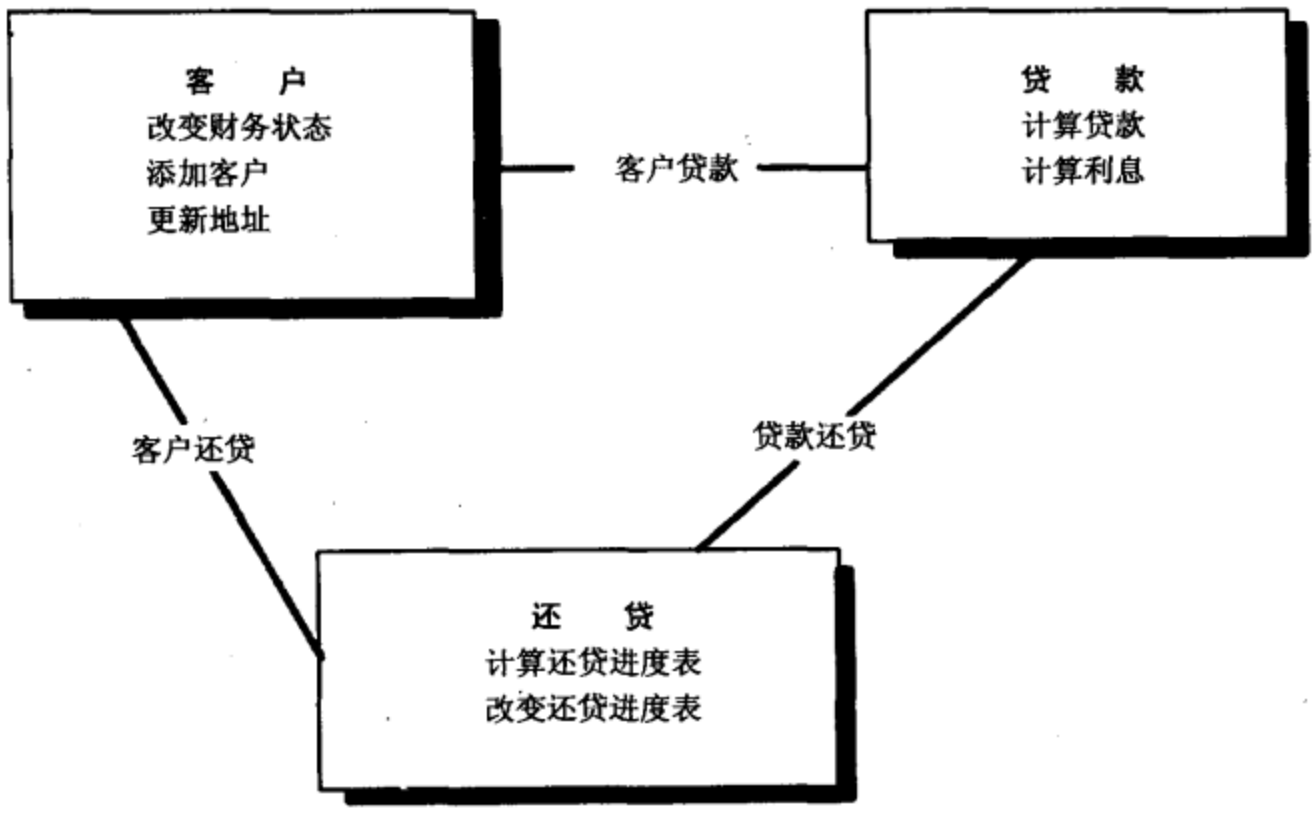



图5-13　客户贷款系统的面向对象模型

### 5.5.8　相关解决方案

如果在存在Functional Decomposition问题的系统中已经投入了过多的工作，你也许可以通过采用类似于The Blob反模式替代方法的办法来挽回事态。

你也许可以把“主例程”类扩展成“协调器”类来管理系统的所有或大部分功能，而不是从头开始重构整个类层次关系。然后可以合并函数类并补充它们，让它们在修改后的“协调器”类指导下进行自己的处理，从而把它们转变成准面向对象的类。这个过程也许可以产生更可用的类
101 层次关系[Fowler 1997]。

### 5.5.9　对其他视角和规模的适用性

架构视角和管理视角在最初预防和在开发进行中监督防止Functional Decomposition反模式方面都扮演了关键的角色。如果一开始就规划了正确的面向对象架构而在开发阶段出现了问题，那么保证实施最初的架构就是管理层面对的管理挑战。与之相似，如果出现问题的原因是最初就普遍缺乏正确的架构，那么发现这个问题、踩下刹车和获取架构性帮助仍然是管理方面的责任——
102 越早进行代价就越低。

反模式

## 5.6 Poltergeist（恶作剧鬼）

**反模式名称**：Poltergeist

**别名**：Gypsy（吉普赛人）[Akroyd 1996]、Proliferation of Classes（类增殖）[Riel 1996]和Big DoIt Controller Class[Fowler 1997]

**最常见规模**：应用层

**重构方案名称**：Ghostbusting（捉鬼）

**重构方案类型**：过程

**根源**：懒惰、无知

**不平衡的力量**：功能管理、复杂性管理

**轶事证据**："我不确定这个类到底做什么，但它肯定很重要！"

### 5.6.1 背景

当Michael Akroyd在1996年的Object World West会议上提出Gypsy反模式时，它把Gypsy类短暂出现然后断续消失的性质比做今天来了明天又走了的"吉普赛大篷车"。在研究Akroyd的模型时，我们希望在该反模式的总体名称中传递有关该Gypsy类的调用函数的更多信息。这样，由于我们觉得既然poltergeist表示"bump-in-the-night（夜里神出鬼没）"的"不停歇的鬼"，这个术语更好地代表了该反模式"突然跳出来造成某些事"的概念，同时又保持了最初的Gypsy名称的"刚才还在这里，突然就消失了"的风味（见图5-14）。

在LISP以及很多其他语言中，总有一些既纯洁又邪恶的程序员，他们非常乐于在系统中利用特定语言函数的"副作用"来完成一些关键功能。分析和理解这样的系统实际上是不可能的，任何对复用的尝试都会被看作是疯狂的举动。 103

与Poltergeist"控制器"类相似，在实现中使用"副作用"来完成任何重要任务都是对语言或架构工具的不正确使用，都应避免。

### 5.6.2 一般形式

Poltergeist类在系统中只承担有限的责任和角色，因此，它们的有效生命周期相当短暂。Poltergeist会建立不必要的抽象，让软件设计变得混乱。它们过度复杂、难以理解而且难以维护。

该反模式通常出现在熟悉过程建模但新接触面向对象设计的设计人员定义架构的时候。在该反模式中，可以发现一个或多个像鬼魂一样虚幻的类，它们只是短暂地出现，用于启动其他更为持久的类中的某些操作。Akroyd称这些类为"Gypsy Wagon（吉普赛大篷车）"[Akroyd 1996]。一般而言，Gypsy Wagon作为控制器类出现，只是按照预定的顺序调用其他类的方法。它们通常 104

很明显，因为它们的名称常常带有_manager或_controller后缀。


图5-14　Poltergeist：鬼类

Poltergeist反模式常常是某些缺乏经验、没有真正理解面向对象概念的架构师故意设立的。Poltergeist类建立的是不良设计制品的关键原因有3个：

（1）它们不是必需的，所以它们每次“出现”的时候都会浪费资源。
（2）它们是低效的，因为它们要使用一些冗余的导航路径。
（3）它们让对象模型产生不必要的混乱，阻碍了正确的面向对象设计。

### 5.6.3　症状和后果

- 冗余的导航路径。
- 瞬时联系。
- 无状态的类。
- 临时的、短持续时间的对象和类。
- 只是为了通过临时联系“播种”或“调用”其他类而出现的单操作类。
- 具有“类似于控制”的操作名称，例如start_process_alpha的类。

### 5.6.4　典型原因

- 缺乏面向对象架构。“设计师不懂面向对象”。
- 使用不正确的工具来完成工作。与普遍观点相反，面向对象方法并不一定是解决所有问题的正确方法。就像一张海报曾经说的：“没有正确的方法来做错事。”意思就是，如果面向对象不是正确的工具，就没有正确的方法来实现它。 105
- 说明灾难。和The Blob反模式一样，管理层有时会在需求分析时做出架构承诺。这是不合适的做法，常常会导致一些问题，这个反模式就是其中之一。

### 5.6.5　已知例外

Poltergeist反模式没有例外。

### 5.6.6　重构方案

Ghostbuster（捉鬼人）解决Poltergeist反模式的方法是从类层次关系中完全移除它们。但是在移除它们之后，由它们“提供”的功能必需被替换。这项工作很简单，只要简单地调整就可以纠正架构。

关键之处是把Poltergeist类最初封装的控制操作移动到它们调用的相关类中。下一部分对此进行了详细解释。

### 5.6.7　示例

为了更清楚地解释Poltergeist反模式，考虑图5-15中制造桃子罐头的例子。我们可以发现PEACH_CANNER_CONTROLLER类是一个Poltergeist类，因为：

- 它到系统中的所有其他类都有冗余的访问路径。
- 它所有的联系都是瞬时的。
- 它没有状态。
- 它是临时的、短持续时间的类，只是为了通过临时联系调用其他类而跳出来。

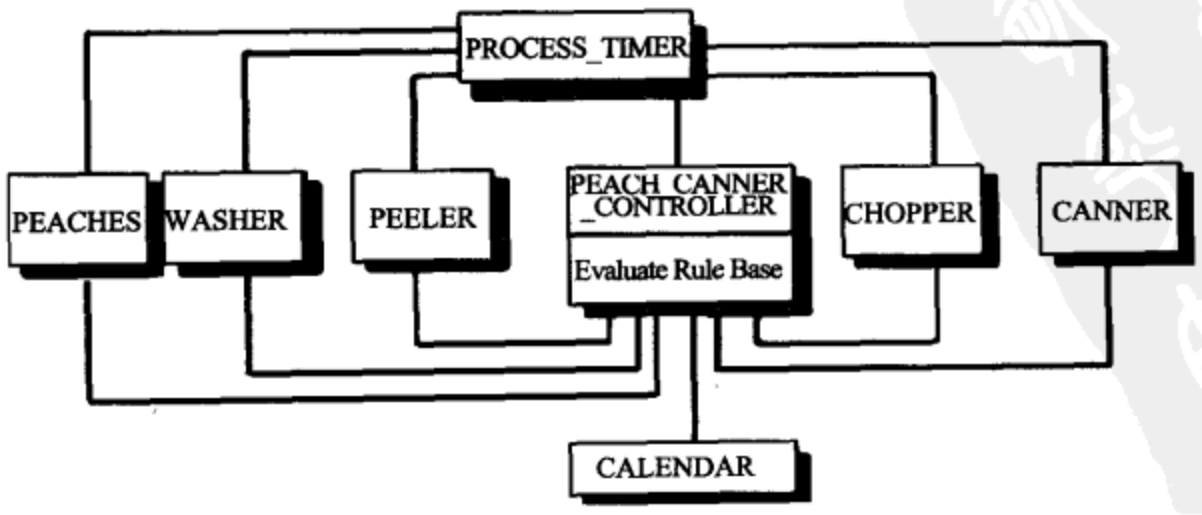



图5-15　控制器类

在这个例子中，如果我们移除Poltergeist类，剩下的类就会失去交互的能力。处理过程不再有任何的顺序。因此，我们要把这种交互能力放置到剩下的层次关系中，如图5-16所示。请注意
106 每步处理都增加了特定的操作，以便各个类可以相互交互并产生结果。

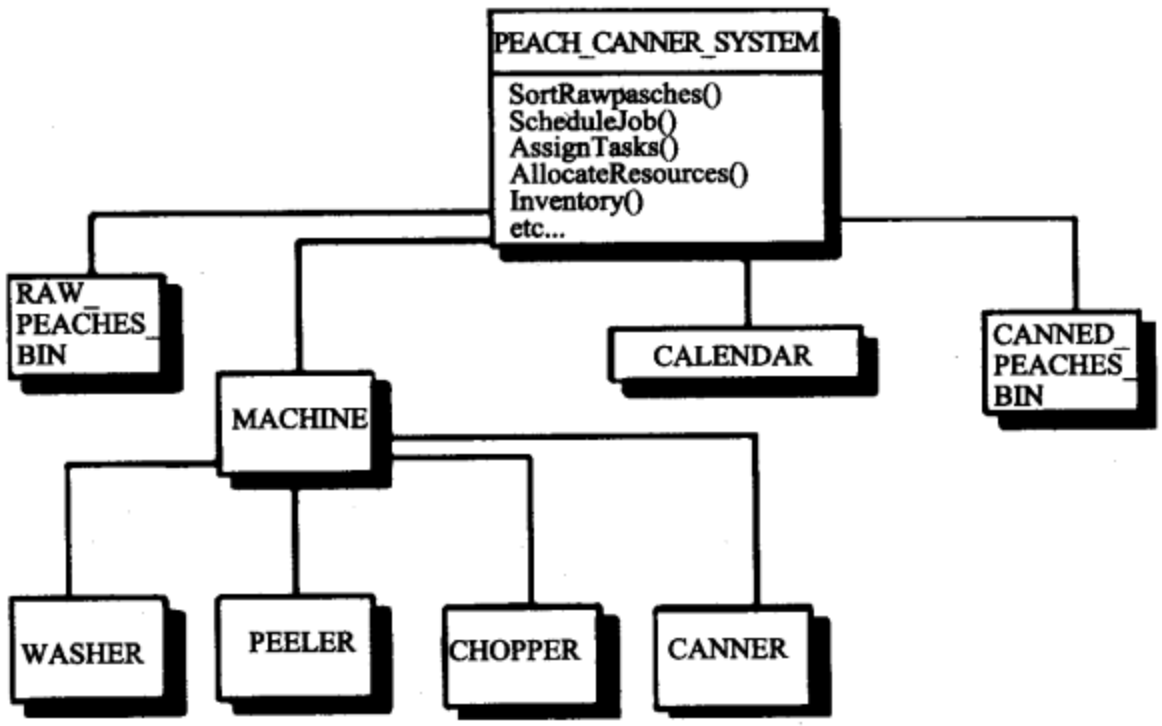



图5-16　重构的控制器类

## 5.6.8　相关解决方案

The Blob反模式中讨论的“80%方案”会产生一些看起来很像Poltergeist的内容。该方法产生的“协调器”类仍然管理系统的所有或大部分功能，通常显现出很多Poltergeist类的特点。

## 5.6.9　对其他视角和规模的适用性

我们选择把这个反模式放在开发性反模式这一章而不是和架构性反模式放在一起，是因为虽然它的确可能是未能正确建立系统架构的结果，但更可能出现在开发人员边实现系统边进行系统设计（通常就是在座位边上设计）的情况下。这是否证明了Poltergeist实际上是失败的管理则取决于读者的判断。

与大多数开发性反模式一样，架构和管理视角在最初预防和持续监督防止它们的方面都扮演了重要的角色。往往通过架构视角才能发现正在出现的反模式，而在未能完全预防反模式时，只
107 有通过有效的管理才能正确解决它。

管理者应该非常注意，保证尽早让具备适当资格的面向对象架构师来评估面向对象架构，然后持续防止新手产生类似于这个反模式的错误。一开始就为良好的架构投资！

### 小型反模式：Boat Anchor（船锚）

**反模式问题**

Boat Anchor是指在当前项目中没有起到有益作用的那些软件或硬件。而且Boat Anchor往往具有相当高的成本，这使得当初对它的采购显得更具有讽刺意味。

获取Boat Anchor的原因在当时往往是非常有说服力的。例如，政策或程序性的关系可能会要求购买和使用特定的硬件或软件。这是该软件项目的启动设想（或限制）。另一个强制性的原因是某个关键的管理者相信获取这些东西的作用。有一种名为“贵宾行销”的销售实践就是把销售目标定位在具有采购权的高级决策制定者。贵宾行销通常把重点放在小型或中型公司的行政总监身上。在进行适当的技术评估之前，他们可能就会做出采购该产品的承诺。

管理者和软件开发人员要面对的后果就是需要投入大量的精力来让采购的产品可以工作。在投入了大量时间和资源后，技术人员认识到这个产品在当前的上下文环境中没有用，于是放弃它而改用别的技术方法。最终，Boat Anchor被扔到某个角落里，渐渐蒙上一层灰尘（如果是硬件的话）。

**重构方案**

良好的工程实践包括提供后备技术，也就是某种替代方法，只需要最少的软件返工就可以把它替换进来。对后备技术的选择是一项重要的风险缓解策略。应该为大部分基础技术（大部分软件所依赖的），以及其他高风险领域中的技术确定相应的后备技术。应该在选择过程中对后备技术和关键途径技术一起进行评估。我们建议对关键途径和后备技术都使用评估许可（大部分供应商都可以提供）来建立原型。

**相关反模式**

在Irrational Management反模式中解释了理性决策制定过程。理性决策制定可以被用做客观的技术选择过程来在采购前发现Boat Anchor。对Smoke and Mirrors反模式的解决方案说明了采购前技术评估的实践方法，包括审查产品文档和购买前培训。

108

109

反模式

## 5.7　Golden Hammer（金锤）

**反模式名称**：Golden Hammer
**别名**：Old Yeller、Head-in-the sand（鸵鸟政策）
**最适用规模**：应用层
**重构方案名称**：Expand your horizons（扩展视线）
**重构方案类型**：过程
**根源**：无知、自负、思想狭隘
**不平衡的力量**：技术转移管理
**轶事证据**："我有一个锤子，所以所有东西就都是钉子。""我们的数据库就是我们的架构。""也许我们根本就不应该使用Excel宏来做这件事。"

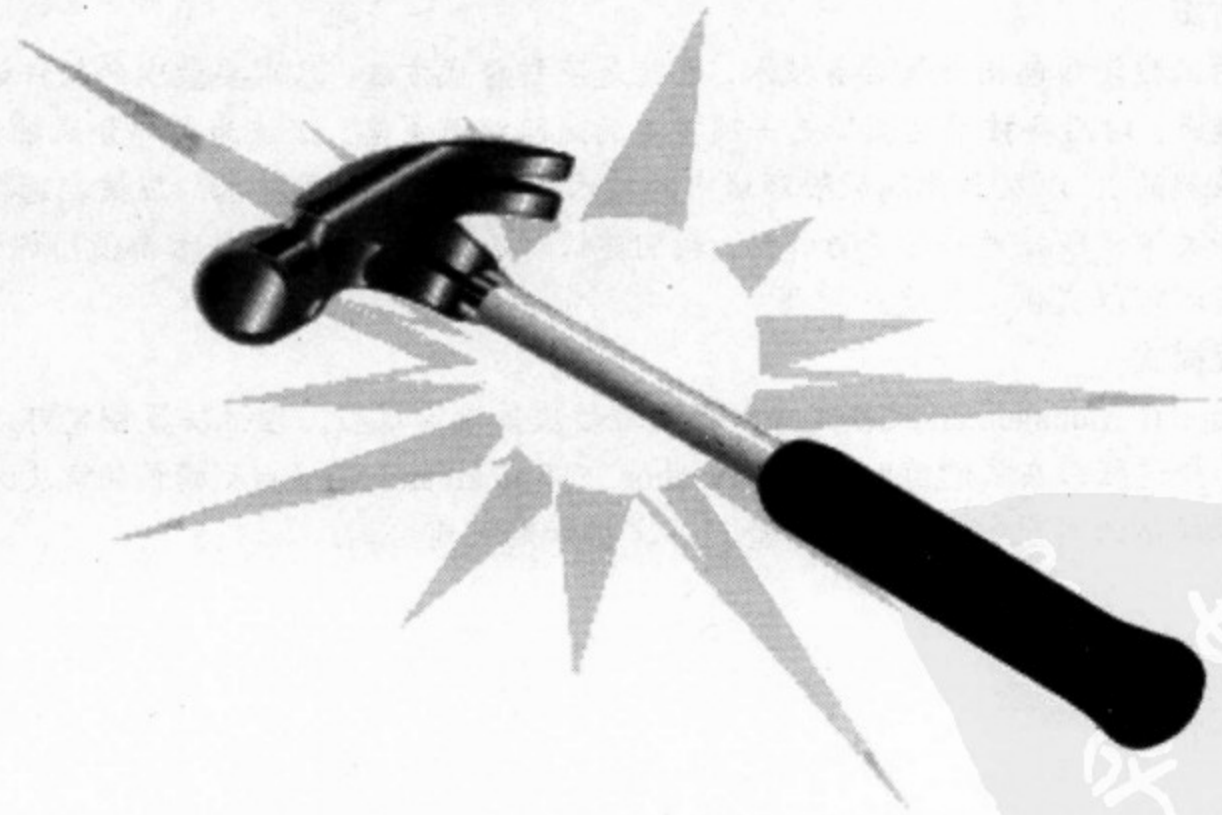


图5-17　当你的工具只有一个锤子的时候，其他所有东西就都是钉子

### 5.7.1　背景

这是软件行业中最常见的反模式之一。供应商，特别是数据库供应商，常常会鼓吹使用他们不断成长的产品套件来解决机构中的大部分需要。考虑到采用特定数据库解决方案的起始成本很高，这些供应商常常会以低得多的价格提供对他们的技术的扩展，这些扩展可以与他们已经部署的产品很好地共同工作。

## 5.7.2　一般形式

如果软件开发团队在使用特定解决方案或特定供应商的产品方面具备了很高的能力，我们就把这种现象称为Golden Hammer。作为其结果，每个新产品或开发工作都被看做最好用它来解决。很多时候，这个Golden Hammer并不适合当前的问题，但团队不会投入很多精力来探索替代解决方案。 111

该反模式会导致错误应用某个受喜爱的工具或概念。开发人员和管理者乐于使用已有的方法，不愿学习和应用更合适的新方法。常见的“我们的数据库就是我们的架构”思维方式就是它的典型表现，在全世界的银行业尤其普遍。

鼓吹者常常会建议使用Golden Hammer及其相关产品套件作为一个机构中大部分问题的解决方案。至于采用特定解决方案的起始成本，Golden Hammer的鼓吹者会争辩说将来对技术的扩展可以让风险和成本最小化，这些扩展会被设计成可以和他们已有的产品共同工作。

## 5.7.3　症状和后果

- 对相当广范围内概念上不同的产品使用相同的工具和产品。
- 与行业中其他解决方案相比，采用的方案在性能和可伸缩性等方面较差。
- 采用特定产品、应用套件或供应商工具集才能最好地描述系统架构。
- 开发人员就系统需求与系统分析师和最终用户进行争论，常常鼓吹容易用特定工具解决的需求，并把他们的关注点从采用的工具不能满足的领域转移开。 112
- 开发人员与行业相隔离。他们表现出对替代方法缺乏知识和经验。
- 为了利用已有的投资而未能完全满足需求。
- 现有产品支配了设计和系统架构。
- 新开发活动严重依赖于特定供应商的产品或技术。

## 5.7.4　典型原因

- 使用相同的方法获得了多次成功。
- 在某个产品或技术的培训上进行了大量投入和/或取得了很多经验。
- 开发群组与行业及其他公司相隔离。
- 依赖于其他行业产品无法马上提供的专有产品特性。
- “Corncob”引发了该问题（参阅第7章中的Corncob反模式）。

## 5.7.5　已知例外

Golden Hammer反模式在下面的情况中有时也是有效的：

（1）定义了架构限制的产品是有目的的长期战略解决方案。例如，将Oracle数据库用于持久存储，使用封装的存储过程进行对数据的安全访问。

（2）产品是满足软件大部分要求的供应商套件的一部分。

### 5.7.6　重构方案

这个解决方案涉及哲学方面以及对开发过程的改变。从哲学角度来看，一个机构需要形成致力于探索新技术的理念。如果没有这种理念，就存在过度依赖特定技术或供应商工具集的潜在危
113 险。这一解决方案要求采取两方面的措施：管理层对开发人员的职业发展做出更多的承诺，以及建立一个开发策略要求明确的软件边界声明来让技术移植成为可能。

需要有良好定义的边界来设计和开发软件系统，这样的边界可以提高单个软件构件的可替换性。构件应该把系统和它的实现的专有特性隔离开。如果采用明确声明的软件边界来开发系统，在构成边界的接口之处就有可能用新的实现来替换当前实现中所使用的软件而不会影响系统中的其他构件。类似于OMG IDL规范的行业标准作为在构件之间建立严格软件边界的工具，其价值是无法衡量的。

此外，软件开发人员需要跟上技术发展的趋势，既包括机构领域内的趋势，也包括整个软件行业中的趋势。通过一些鼓励技术思想交流的活动可以实现这一目的。例如，开发人员可以建立群组来讨论可能会在将来影响到所在机构的技术发展（设计模式、正在形成的标准和新产品）。他们还可以成立读书研讨俱乐部（书研会）来追踪和讨论说明软件开发新方法的出版物。我们在实践中发现书研会是交流观点和新方法时非常有效的办法。即使没有管理层的全面支持，开发人员也可以与有技术思想的人建立非正式的网络，来研究和跟踪新技术和解决方案。行业会议也是一种很好的方法，可以接触其他人和供应商，保持对行业走向以及开发人员可以使用哪些新方案的了解。

对管理层而言，另一个有助的步骤是建立采用开放式系统和架构的决心。如果缺乏这种决心，开发人员往往会认为为了达到短期目标而采取各种必要手段都是可取的。虽然在短期来看这种做法也许是可以令人满意的，但是其未来的结果可能会出现问题，因为需要花大量的精力来对过去的软件进行返工以便适应新的挑战，而不是把系统建立在由过去的成功所形成的坚实基础上。灵活的、可复用的代码要求在最初开发时就进行投入，否则就无法获得长期的效益[Jacobson 1997]。而且，在产品或项目开发中过分依赖特定技术或供应商工具集是一种潜在的风险。在测试如何把风险较低的开放式技术结合到开发工作中来这个方面，开发概念验证原型的内部研究程序相当有效。

114 另一个在管理层消除或避免Golden Hammer反模式的方法是鼓励雇用来自不同领域、具有不同背景的人。在开发解决方案时，团队可以从具有更广泛的经验基础中受益。与一支对相当广泛的数据库技术方案很有经验的团队相比，所有成员都是对同一个数据库产品具有经验的团队极大地限制了可能的解决方案的空间。

最后，管理层必须在软件开发人员的职业发展方面积极投入，并奖励那些主动改进自身工作的开发人员。

### 5.7.7　变化

Golden Hammer的一个常见变化是开发人员特别喜欢使用某个软件概念。例如，有些开发人员学到了一两个GoF模式[Gamma 1994]，就把它们应用到所有的软件分析、设计和实现中。他们找出可以套上这些设计模式结构的地方，并强行把它们应用在整个开发过程中，对模式意图和目的的讨论都不能动摇他们。只有通过教育和指导才能帮助大家认识到软件系统构造的其他可用方法。

### 5.7.8　示例

Golden Hammer反模式的一个常见例子是在以数据库为中心的环境中，除了数据库供应商提供的架构之外就没有别的架构。这样的环境中，在开始面向对象分析之前就已经设定了要使用特定的数据库。这样，软件生命周期常开始于建立E-R（实体—关系）图作为客户提出的需求文档。这种做法往往是破坏性的，因为该E-R图最终被用于说明数据库需求；而在理解系统和建模之前详细设计子系统结构的做法假定了实际客户的需求对系统设计没有什么影响。需求收集活动应该让系统开发人员可以理解用户的需要，直到结果系统的外部表现可以被用户看作是黑盒[Davis 1993]。可以相信的是，有许多系统根本不使用数据库就可以满足用户需求。但是，如果Golden Hammer反模式在起作用，这种可能性从一开始就被降低了，导致每个问题都需要使用数据库才能解决。 115

随着时间的过去，机构可能会把一些本来可以实现成独立系统的产品开发成以数据库为中心的。数据库发展成应用间互联的基础，它管理对数据的分布式和共享式的访问。此外，利用数据库的专有特性解决了很多实现问题，而这些专有特性都承诺将来会提供移植以与该实现数据库的技术发展保持同步。可是到了某些时候，它可能要与其他系统进行互用，这些系统要么没有使用相同的以数据库为中心的架构，要么使用了不允许对其信息进行不受限制访问的不同数据库。突然之间，开发变得极其昂贵，因为要在不同系统之间建立独特的、专用的连接“桥梁”。但是，如果在情况无法控制之前就考虑到这个问题，就可以建立一个共同的框架，根据标准接口规范如CORBA、DCOM或TCP/IP来选择用于特定领域的产品。

另一个例子是一个具有数个烟囱遗留系统的保险公司，它决定要转换到客户机/服务器模式，使用Microsoft Access作为持久存储的关键部分。该公司的呼叫中心系统的整个前端架构都是围绕着该产品的一个早期版本。由于一个拙劣的架构决策，其后系统的未来就完全受制于该数据库产品的开发途径。不用说，该系统只维持了不到6个月。

### 5.7.9　相关方案

- Lava Flow。如果Golden Hammer反模式在好几年间被应用于很多项目，就会导致Lava Flow反模式。一般来说，根据Golden Hammer的早期版本建立的较老部分就代表了整个应用中那些很少被使用的部分。开发人员不愿意修改这些部分。它们随时间而堆积起来，增加

了应用的总体规模，但是只实现了用户很少使用甚至完全不会用到的功能。

- Vendor Lock-In。当开发人员主动接受供应商对应用Golden Hammer反模式的支持和鼓励时，就产生了Vendor Lock-In。也就是软件项目在设计和实现面向对象系统时主动承诺依赖于单个供应商提供的方法。

116

## 小型反模式：Dead End（死胡同）

### 别名

Kevorkian Component

### 反模式问题

如果对一个可复用构件进行修改后它不再受供应商的支持，那么对它的修改就会形成Dead End。做出这些修改后，提供支持的负担就会转移到应用系统开发人员和维护人员身上。对可复用构件的改进将难以被集成，而且如果出现支持问题，也会被归咎到这些修改。

构件的提供者也许是一个商业供应商，这时该反模式也被称为COTS Customization（COTS定制）。当该产品出现后续版本时，如果还有可能，就要再次做出这些特别的修改。实际上，由于类似成本和人员更替之类的多种原因，也许就不可能升级定制的构件。

系统集成者做出修改可复用构件的决策，通常被看作是为了绕开供应商产品的不足。作为一种短期手段，它有助于产品开发的进程而不是减慢它。而在考虑到应用的将来版本和“可复用构件”供应商的新版本时，长期的支持负担则是难以维持的。我们惟一一次看到这种做法能够起作用的时候，是系统集成者与可复用构件供应商一起安排在供应商产品的下一个版本中包含SI修改。他们具有相同的目标完全是一种运气。

### 重构方案

117

要避免COTS Customization和对可复用软件的修改。使用主流平台和COTS基础结构，并按照供应者的发布进度来进行升级可以把出现Dead End的风险最小化。如果无法避免进行定制，就使用一个隔离层（参阅Vendor Lock-In反模式）。使用隔离层和其他方法把来自应用软件主体的依赖和定制及专用接口隔离开。

在类似用后抛弃代码的用于支持基础研究的测试床上，Dead End可能是可以接受的解决方案，通过定制可以获得显著的效益。

118

反模式

## 5.8　Spaghetti Code（面条代码）

**反模式名称**：Spaghetti Code
**最常见规模**：应用层
**重构方案名称**：Software Refactoring（软件重构）、Code Cleanup（代码清理）
**重构方案类型**：软件
**根源**：无知、懒惰
**不平衡的力量**：复杂性管理、变化管理
**轶事证据**："啊！真乱哪！""你确实知道这种语言支持不只一个函数，不是吗？""重写这段代码比试着修改它更容易。""软件工程师不写Spaghetti Code。""你的软件结构的质量是为将来的修改和扩展进行的投资。"

### 5.8.1　背景

Spaghetti Code反模式是经典的，也是最著名的反模式。从编程语言的发明之日起，它就以这样或那样的形式出现了。非面向对象语言似乎更容易受该反模式的影响，但在那些尚未完全掌握面向对象所蕴含的高级概念的开发人员当中，这个反模式也相当常见。

### 5.8.2　一般形式

Spaghetti Code就是看上去几乎没有软件结构的一段程序或一个系统。编码和逐步的扩展严重破坏了软件的结构，以至于即使对它的原始开发人员来说，如果他离开该软件一段时间，也会弄不清系统的结构。如果是使用面向对象语言进行开发，该软件可能会包含少量对象，它们的方法具有非常庞大的实现，调用一个单独的、多阶段的处理流程。而且，对对象方法调用的可预测性很高，系统中对象之间的动态交互少到可以忽略的程度。很难维护和扩展该系统，也没有机会复用其他相似系统中的对象和模块。

119

### 5.8.3　症状和后果

- 在代码挖掘后，可以发现只有部分对象和方法看起来适于复用。挖掘Spaghetti Code的回报相对于投入常常相当可怜。在做出进行挖掘的决策之前就要考虑到这一点。
- 方法是面向处理过程的；实际上，对象常常被按照处理过程命名。
- 执行流由对象的实现控制，而不是由对象的客户控制。
- 对象间只有最小限度的关系。
- 很多对象方法没有参数，使用类变量或全局变量进行处理。
- 对象的使用模式具有很强的可预测性。

- 代码难以复用，进行复用的方式往往是通过复制代码。很多时候，根本就没有考虑过代码复用。
- 难以留住那些具有面向对象才能的人。
- 失去了面向对象带来的益处。没有使用继承来扩展系统；没有使用多态机制。
- 后续维护工作会加剧这个问题。
- 软件迅速达到回报渐小点；维护现有代码集的成本比从头开始开发新方案的成本更高。

“先做重要的事，不重要的就不要做。”

——Shirley Conran（英国女作家）

### 5.8.4　典型原因

- 对面向对象设计技术缺乏经验。
- 没有适当的指导；失效的代码评审。
- 实现前未进行设计。
- 通常是开发人员孤立工作的产物。

120

### 5.8.5　已知例外

在接口是连贯的而只有实现是面条形式时，Spaghetti Code反模式在一定程度上是可以接受的。这就像封装一段非面向对象的代码。如果该构件的寿命很短，而且清晰地与系统其他部分隔开，那么可以忍受一定量的低质代码。

软件行业的现实就是对软件质量的考虑通常要让位于商业考虑，而某些时候，商业上的成功取决于尽快交付某个软件产品。如果软件架构师和开发人员不熟悉要处理的领域，可能较好的做法是先开发（质量一般的）产品来获得对领域知识的了解，目的是晚些时候再采用改进的架构来设计产品[Foote 1997]。

### 5.8.6　重构方案

软件重构（或代码清理）是软件开发中的重要部分[Opdyke 1992]。超过70%的软件成本都是由扩展造成的，所以维持一个支持扩展的连贯软件结构非常关键。当为了支持预料之外的需求而破坏了结构时，代码支持扩展的能力就受到了限制，直到最终不复存在。不过，“代码清理”这个术语对那些高层管理者没有吸引力，所以最好换个类似“软件投资”之类的词来讨论这个问题。毕竟，从非常实际的角度来说，代码清理是对软件投资的维护。结构良好的代码会有更长的生命周期，可以更好地支持业务领域和内在技术的变化。

理想情况下，代码清理应该是开发过程的自然组成部分。在代码中每增加一个功能（或一组功能），就应该接着进行代码清理来恢复或改进代码结构。根据增加新功能的频率，代码清理可以按照小时或天来进行。

代码清理还支持对性能的改善。性能优化一般遵循90/10规则，也就是要达到最优性能的90%，只需要对10%的代码进行修改。对单子系统编程或应用编程，性能优化常常会造成在代码结构上的折中。第一步的目标是获得令人满意的结构；然后是通过测量确定对性能很关键的代码的存在位置；最后是谨慎地引进必要的结构折中来改善性能。有时必须为了重要的系统扩展而取消软件中为了改善性能而做出的变化。为了给将来的版本保留这些软件结构，对这些地方需要进行额外的文档说明。 121

解决Spaghetti Code反模式的最佳方法是进行预防；也就是说先思考，然后在编写代码前建立一个行动计划。但是，如果代码集已经退化到无法维护的地步，而且软件重新设计也不可行，还是可以采取一些其他步骤来避免问题的恶化。首先，在维护过程中向Spaghetti Code代码集中加入新功能时，不要用类似于刚好满足新需求的风格来修改代码，而是总是花一些时间来把已有系统重构到更可维护的形式。软件重构包括对已有代码进行下列操作：

（1）使用访问函数来获得对类中成员变量的抽象访问。编写新的和重构的代码时使用这些访问函数。

（2）把一段代码转换成可以在将来的维护和重构工作中复用的函数。抵制Cut-and-Paste反模式（接下来将讨论）的诱惑至关重要。应该使用Cut-and-Paste的重构方案来修补以前实现的Cut-and-Paste反模式。

（3）重新排列函数的参数，从而在整个代码集中获得更高的一致性。即使是不好的但是一致的Spaghetti Code，也比不一致的Spaghetti Code好维护。

（4）移除将会或已经无法访问的代码。反复地未能发现和移除过时代码是造成Lava Flow反模式的主要原因。

（5）重命名类、函数或数据类型来遵守企业或行业标准，符合可维护实体的要求。大部分软件工具都支持全局重命名。

简而言之，一旦需要修改代码集，就在资源许可范围内致力于积极地重构和改进Spaghetti Code。单元测试和系统测试工具及应用软件的使用，对保证重构没有给代码集立即产生任何新缺陷非常有效。经验表明，软件重构带来的效益远远超过了额外的修改可能产生新缺陷的风险。 122

如果可以选择预防Spaghetti Code，或者如果你可以完全重新设计一个Spaghetti Code应用，可以采取下面这些预防性手段：

（1）不管对领域的理解程度如何，坚持采用正确的面向对象分析过程来建立领域模型。对任何中等规模或大规模的项目来说，建立一个领域模型作为设计和开发的基础都是至关重要的。如果对领域理解得非常充分，让人觉得不需要领域模型了，可以用“如果是那样的话，建立一个模型的时间也就是可以忽略的”来反驳。如果这个时间确实可以忽略，就礼貌地承认你开始时错了。否则的话，花掉的时间就足以证明它的重要性了。

（2）在建立了领域模型来解释系统需求及要解决的可变性范围之后，建立一个隔离的设计模型。虽然使用领域模型作为设计的起点也是有效的；但必须把领域模型保持原样以保留那些有用

的信息。否则，如果允许它直接发展成设计模型，就会失去那些信息。设计模型的用途是抽取领域对象之间的共同性，通过抽象来阐明系统中必需的对象和关系。如果正确进行，它会建立软件实现的边界。实现只应用于满足系统需求，这些需求或者是由领域模型明确指出的，或者是系统架构师或高级开发人员所预期的。

（3）建立设计模型时，重要的是保证把对象都分解到可以被开发人员完全理解的层次。要让开发人员而不是设计人员相信软件模块易于实现。

（4）一旦对领域模型和设计模型都进行了第一遍设计，就可以根据设计建立的计划开始实现。
123 设计不必是完整的，其目的只是在于总是应该根据某个预定计划来进行软件构件的实现。开始开发以后，继续以增量的方式检验领域模型的其他部分，并设计系统的其他部分。随着时间的流逝，可以精化领域模型和设计模型，以接纳需求收集中的发现和设计决策，解决实现方面的问题。再次强调，如果有整体软件开发过程，在实现前说明需求和设计而不是让它们同时发生，那么出现Spaghetti Code的可能性可以小得多。

### 5.8.7　示例

这是新接触面向对象软件开发的人会表现出的一个常见问题，他们把系统需求直接映射成函数，使用对象作为组合相关函数的地方。每个函数都包含完整实现特定任务的整个处理流。例如，下面的代码段包含类似initMenus()、getConnection()和executeQuery()的函数，它们完整地执行了所说明的操作。每个对象方法都包含了独立的处理流，按照完成该任务所需的顺序执行所有的步骤。对象在连续调用之间只保留很少的状态信息，或者根本没有。类变量只是用于临时保存单个处理流的中间结果。

**代码清单5-1**

```
public class Showcase extends Applet
implements EventOutObserver {
//Globals
String
homeUrl="http://www.webserver.com
/images/"
;

int caseState;
String url="jdbc:odbc:WebApp";
Driver theDriver;
Connection con=null;
ResultSet rs,counter;
int theLevel;
int count=0;
String tino;
int [] clickx;
int [] clicky;
String [] actions;
String [] images;
String [] spectra;
String showcaseQuery=null;
TextArea output=null;
Browser browser=null;
Node material=null;
EventInSFColor diffuseColor=null;
EventOutSFColor outputColor=null;
EventOutSFTime touchTime=null;
boolean error=false;
EventInMFNode addChildren;
Node mainGroup=null;
EventOutSFVec2f coord=null;
EventInSFVec3f translation=null;
EventOutSFTime theClick=null;
Image test;
int rx,ry;
```

```
float arx,ary;
int b=0;
Graphics gg=null;

//Initialize applet
public void init() {
  super.init(); setLayout(null);
  initMenus();
  output=new TextArea(5, 40);
  add(output);
  browser=(Browser)
Browser.getBrowser((Applet)this);
  addNotify(); resize(920,800);
initUndoStack();
   caseState=0; theLevel=0;
   setClock(0);
try { theDriver=new postgresql
.Driver(); }
catch(Exception e) {};
try { con=DriverManager
.getConnection(

"jdbc:postgresql://www.webserver
.com/WebApps",
     "postgres","");
     Statement stmt=con
     .createStatement();
     showcaseQuery="SELECT sid,
     case,
button,text , name, actions FROM
WebApp
WHERE case="+caseState+" and
level="+theLevel+";";
      rs=stmt.executeQuery
     (showcaseQuery);
      count=0; while (rs.next())
      count++;
      System.out.println("Count=
      "+count+"\n");
      rs=stmt.executeQuery
      (showcaseQuery);
}
catch(Exception e) {System.out
.println(
      "Error connecting and running:
      "+e);};
   nextButton=new
symantec.itools.awt.ImageButton();
  lastButton=new
symantec.itools.awt.ImageButton();
  try {
     nextButton.setImageURL(new
java.net.URL(

"http://www.webserver.com:8080/
images/next.jpg"));
      if (count<7) nextButton
      .setVisible(false);
      else nextButton.setVisible
     (true);
      lastButton.setImageURL(new
java.net.URL(

"http://www.webserver.com:8080/
images/last.jpg"));
  }
  catch(Exception e) {};
imageButtons=new
symantec.itools.awt.ImageButton[6];
11=new
symantec.itools.awt.shape.Horizontal
Line();
12=new
symantec.itools.awt.shape.Horizontal
Line();
v1=new
symantec.itools.awt.shape.Vertical
Line();
v2=new
symantec.itools.awt.shape.Vertical
Line();
bigspectralabel=new
java.awt.Label("Spectra");
gtruthlabel=new
java.awt.Label("GroundTruth");
clickx=new int[6];
clicky=new int[6];
actions=new String[6];
images=new String[6];
spectra=new String[6];
```

```
imageLabels =new java.awt.Label[6];
for (int I=0; i<6 ; i++) {
  imageButtons[i]=new
symantec.itools.awt.ImageButton();
  imageLabels[i]=new java.awt
  .Label();
  actions[i]=new String();
  images[i] =new String();
  spectra[i] =new String();

 };
for (int i=0; i<6 ; i++) {
 try{
   rs.next();
   tino=rs.getString(4);
System.out.println(tino+"\n");
    actions[i]=rs.getString(6);}
   catch(Exception e) {System.out
   .println("SQL
Error :"+e);}
   try{
      System.out.print(tino+ln"\n");
   int len=tino.length();
   if (tino.startsWith
   ("INVISIBLE")) {
     imageButtons[i]
     .setVisible(false);
     imageLabels[i]
     .setVisible(false);}
 else {
 imageButtons[i].setImageURL(
     new java.net.URL
     (homeUrl+tino));
 imageButtons[i].setVisible
 (true);
 imageLabels[i].setText
 (rs.getString(5));
 imageLabels[i].setVisible
 (true);
 }
} catch (Exception e) { System.out
.println(
     "Died in accessor statement:
     "+e);
     }
}
11.reshape(0,6,775,1);add(11);
12.reshape(0,120,775,1);add(12);
v1.reshape(0,6,1,114);add(v1);
v2.reshape(775,6,1,114);add(v2);
bigspectralabel.reshape
(460,122,200,16);
bigspectralabel.setVisible(false);
gtruthlabel.reshape(124,122,200,16);
gtruthlabel.setVisible(false);
add(bigspectralabel);add
(gtruthlabel);
nextButton.reshape(2,12,84,40);
add(nextButton);
lastButton.reshape(2,56,84,40);
add(lastButton);
imageLabels[0].reshape(124,12,84,16);
add(imageLabels[0]);
imageButtons[0].reshape(124,30,84,84);
add(imageButtons[0]);
imageLabels[1].reshape
(236,12,84,16);
add(imageLabels[1]);
imageButtons[1].reshape
(236,30,84,84);
add(imageButtons[1]);
imageLabels[2].reshape
(348,12,84,16);
add(imageLabels[2]);
imageButtons[2].reshape
(348,30 ,84,84);
add(imageButtons[2]);
imageLabels[3].reshape
(460,12,84,16);
add(imageLabels[3]);
imageButtons[3].reshape
(460,30 ,84,84);
add(imageButtons[3]);
imageLabels[4].reshape
(572,12,84,16);
add(imageLabels[4]);
imageButtons[4].reshape
(572,30 ,84,84);
add(imageButtons[4]);
imageLabels[5].reshape
```

```
(684,12,84,16);
add(imageLabels[5]);
imageButtons[5].reshape
(684,30 ,84,84);
add(imageButtons[5]);
// Take out this line if you don't
use
// symantec.itools.net.RelativeURL
symantec.itools.lang.Context
.setDocumentBase(
  getDocumentBase());
 //{{INIT_CONTROLS
 //}}
}
```

124
~
126

### 5.8.8　相关解决方案

- Analysis Paralysis。该反模式是把Spaghetti Code的解决方案带到了逻辑极限的结果。它不是在没有设计来指引代码的整体结构时即兴开发代码，而是产生一个详细的设计，却没有地方可以着手开始实现。
- Lava Flow。该反模式常常包含多个Spaghetti Code的样本，阻碍对已有代码集的重构。在Lava Flow中，代码集在生命周期中的某些时候具有特定的逻辑目的，但是其中的某些部分虽然已经过时，却仍然保留在代码集中。

#### 小型反模式：Input Kludge（输入拼凑）

**反模式问题**

未能通过直接行为测试的软件可能就是Input Kludge的例子。在采用即兴实现的算法（ad hoc algorithm）[①]来处理程序输入的时候就有可能发生这种反模式。例如，如果程序接受用户的自由文本输入，即兴实现的算法可能会错误处理很多合法的和非法的输入字符串组合。关于Input Kludge有一种有趣的说法："最终用户只要接触键盘一会儿就能让新程序完蛋。"

**重构方案**

对于非演示用途的软件，应使用具有产品质量的输入处理算法。例如，词法分析和解析软件是很容易获得的自由软件。类似于lex和yacc的程序可以稳健地处理由正则表达式和上下文无关的语法构成的文本。我们建议采用这些技术来建立具有产品质量的软件，以保证正确处理非预期的输入。

127

**变化**

许多软件缺陷都是由于用户可访问功能的非预期组合造成的。我们建议在具有图形用户界面的复杂程序中使用功能矩阵。功能矩阵就是在程序中用于在用户操作之前启用或禁用某些功能的状态信息。当用户调用一个功能的时候，功能矩阵将指出需要禁用哪些其他功能，以避免出现冲突。例如，在显示菜单之前常会使用功能矩阵来突出显示或非突出显示某些菜单命令。

**背景**

程序员受到培训要避免会导致程序和系统崩溃的输入组合。在一次有关OpenDoc的实验培训课上，我们使用了该技术的一个Alpha版，该版本对于产品质量的开发来说还不够稳健。也

① 指自行临时编写的处理算法，与已经验证的（如lex与yacc）相对。——编者注

就是说，它很容易通过一些看起来正确的输入命令和鼠标操作序列让整个操作系统崩溃。学生们的第一天几乎都花在无数次的系统崩溃和等待系统重启上。在体验了这个“崩溃实验室”后，我们很怀疑该版本的稳健程度是否能支持进行任何形式的复杂软件开发。不过到那一周结束的时候，我们已经学会了如何绕过这些限制完成编程任务和输入操作，这些远远超过了我们在第一天形成的期望。我们已经被这些能够避免系统崩溃的输入序列潜移默化了。

128

## 小型反模式：Walking through a Mine Field（穿越雷区）

### 反模式问题

使用当今的软件技术就像是穿越高科技雷区[Beizer 1997a]（见图5-18）。该反模式也被称为Nothing Works（没什么能工作）或Do You Believe Magic（你相信魔法吗）？发布的软件产品中有无数的错误；实际上，专家估计在原始代码中每行代码包含2～5个错误。这意味着需要对每行代码进行两处以上的改动才能去除所有错误。毫无疑问，许多产品在发布的时候还远不能支持可运行的系统。一位有见识的软件工程师说过：“没有真正的系统，即使我们的也不是。”


图5-18　穿过雷区的最好办法就是跟着别人走

129

软件缺陷的位置和后果与它们的表面原因无关，即使一个微小的错误也可能会产生灾难性后果。例如，操作系统（UNIX和Windows等）含有许多已知或未知的安全缺陷，让它们易于受到攻击。而且，因特网显著增加了系统攻击的可能性。

最终用户常常会遇到软件错误。例如，大约有1/7正确拨号的电话呼叫没有被电话系统（一种软件密集的应用）正确完成。而且请注意，与软件失败的频率相比，用户抱怨的比例是相当低的。

商业软件测试的目的是限制风险，尤其是限制支持工作的成本[Beizer 1997a]。对于简装软件产品，每当一个最终用户联系供应商以获取技术支持，他的大部分甚至全部利润就被答复电话花掉了。

相比过去较简单的系统，我们是很幸运的。出现软件错误的时候，最可能的结果就是什么

都没有发生。而对于当今的系统，包括计算机控制的载客火车和太空飞船控制系统，错误的后果会是灾难性的。已经有超过半打的重大软件失败导致了超过1亿美元的经济损失。

**重构方案**

通过在软件测试中的适当投入才能让系统相对而言没有错误。在某些领先的公司中，测试人员的数量超过了编程人员[Cusumano 1995]。对测试过程的最重要改变是对测试用例的配置控制[Beizer 1997a]。典型的系统要求的测试用例软件会是产品软件的5倍。测试软件常常比产品软件更复杂，因为对很多错误的检测都需要对执行时机进行明确的管理。测试软件检测到一个错误时，这个错误更有可能来自测试本身而不是来自被测试的代码。配置控制让对测试软件资产的管理成为可能；例如，可以支持回归测试。

测试的其他有效方法包括对测试执行过程和测试设计的自动化。手工执行测试是劳动密集型的工作，手工测试也没有经过验证的依据。与之相反，自动执行测试可以让测试的运行与构建周期保持一致。可以在没有手工干预的条件下进行回归测试，保证对软件的修改不会在以前测试过的行为中产生缺陷。测试设计自动化支持生成严格的测试套件，有数十种很好的工具可以支持测试设计的自动化。

**变化**

有一些应用使用形式验证来保证设计没有错误。形式验证包括校对（以数学方式）对需求的满足程度。不过，受过培训来进行这种类型的分析的计算机科学家相对稀少。此外，形式验证的成本很高，而且结果可能是主观的。因此，我们通常对大部分机构都不推荐它作为可行的方法。

软件评审是一种替代方法，在相当大范围的各种机构中都显示出是有效的[Gilb 1995]。软件评审是审查代码和文档产品的正规过程。它要求仔细审查软件文档来寻找缺陷；例如，它建议每个评审员对每页文档花大约45分钟来寻找缺陷。然后，在评审记录会议上列出多个评审员发现的缺陷。文档编辑可以移除这些缺陷，供评审组进行后续的审阅。评审组为初步接受文档和完成评审过程建立质量标准。软件评审是一个特别有用的过程，因为它可以应用于开发的任何阶段，从编写最初需求文档到编码时都可以。

**背景**

“你相信魔法吗？”这是聪明的计算机专业人员有时会提出的问题。如果你相信当今的软件系统是稳健的，那么你一定相信魔法。

当今软件技术的现实与Stephen Gaskin的*Mind at Play*[Gaskin 1979]中的一个有趣的小故事很相似。在那个故事中，人们开着闪亮的新车，过着舒适的生活。然而，有一个人想看看世界的真实样子。他找到一个权威人物来消除他感觉中的所有幻觉。然后，当他再次看这个世界的时候，他看到人们都走在街上，假装开着华丽的车子。也就是说，那种豪华生活方式是虚假的。最后，这个人放弃了，要求还原到他觉醒之前的状态。

从某个角度来看，当今的技术非常类似于Gaskin的故事。我们很容易相信我们正在强大稳健的平台上使用成熟的软件技术。实际上，这是一个幻想。软件错误非常普遍，而且也没有稳健的平台来承载它们。

反模式

## 5.9　Cut-And-Paste Programming（剪贴编程）

**反模式名称**：Cut-and-Paste Programming

**别名**：Clipboard Coding（剪贴板编码）、Software Cloning（软件克隆）、Software Propagation（软件繁殖）

**最常见规模**：应用层

**重构方案名称**：Black Box Reuse（黑盒复用）

**重构方案类型**：软件

**根源**：懒惰

**不平衡的力量**：资源管理、技术转移管理

**轶事证据**："嘿，我以为你已经修复了那个错误，为什么它又这样做？""伙计，你工作得真快。3周内完成40万行代码是了不起的进展！"

"不要管质量，看看它的宽度。"

——Vince Powell和Harry Driver（英国喜剧作家）

### 5.9.1　背景

Cut-And-Paste Programming是软件复用中非常常见但却是退化的形式，会造成维护的恶梦。它来自于更容易修改已有软件而不是从头编写的观念。通常这是对的，代表了良好的软件开发本能。但是，也很容易过度使用这一方法。

### 5.9.2　一般形式

133 通过在软件项目的多个地方发现相似的代码段可以鉴别该反模式。通常，项目中会有许多程序员正在通过跟随更有经验的开发人员的示例来学习如何开发软件。然而，他们学习的方法是修改那些已经证实可以在相似情况下工作的代码，潜在地定制它来支持新数据类型或稍微改变了的行为。这会造成重复的代码，可能会具有一些短期的积极效果，例如提高用于对工作成绩进行度量的代码行数量值。而且，由于开发人员对在他的应用中使用的代码具有完全的控制，可以很容易扩展这些代码，迅速实现用来满足新需求的短期修改。

### 5.9.3　症状和后果

- 虽然做了很多局部修正，同样的软件错误仍反复出现于软件的多个地方。
- 代码行数量增加了，但整体生产率并没有提高。

- 代码审阅和评审不必要地扩展。
- 难以定位和修正特定错误的所有实例。
- 代码被认为是不需解释的。
- 复用代码时只做了最少的工作。
- 该反模式导致过多的软件维护成本。
- 软件缺陷在整个系统中被复制。
- 可复用的软件资产没有被转换成易于复用和文档化的形式。
- 开发人员为错误建立多个独特的修正，无法从这些变化形式建立标准的修正方法。
- Cut-and-Paste Programming形式的复用让开发的代码行数量虚假地增加，但是不能像其他形式的复用一样降低维护成本。

### 5.9.4　典型原因

- 建立可复用的代码要付出大量的努力，而开发机构强调短期盈利而不是长期投资。
- 没有和代码一起保留软件模块的上下文环境或示意图。
- 开发机构不提倡或不奖励可复用构件，而且对开发速度的要求掩盖了所有其他的评估因素。
- 开发人员缺乏抽象的能力，往往伴随着对继承、合成和其他开发策略的贫乏理解。
- 开发机构强调代码必须完全匹配新任务才能进行复用。为了满足被认为是（事实上也许并不是）独特问题集的需要，复制代码来解决假想的不足。
- 可复用构件在建立之后没有被充分地文档化，或者开发人员并不容易获得它。
- 在开发环境中有“不是在这里发明的”综合症的作用。
- 开发团队缺乏远见或前向思维。
- 在开发人员不熟悉新技术或新工具时，容易发生Cut-and-Paste反模式。由于他们不熟悉，所以会拿过一个可以工作的例子，修改或调整它来满足特定需要。

### 5.9.5　已知例外

在只有尽快放出代码这惟一的目标时，Cut-and-Paste Programming反模式是可以接受的。不过，其代价就是维护成本的增加。

### 5.9.6　重构方案

在白盒复用是系统扩展的支配形式的开发环境中，常常会发生软件克隆。在白盒复用中，开发人员主要通过继承来扩展系统。毫无疑问，继承是面向对象开发的重要部分，但是它在大型系统中存在一些缺点。首先，给一个对象建立子类进行扩展要求对该对象的实现方式，例如被继承基类指定的限制和使用模式，具有一定的了解。大部分面向对象语言都只有很少的限制，派生类

中可以实现各类扩展，导致对子类的非最优使用。此外，一般而言，只在应用编译时（对编译语言而言）才有白盒复用的可能，因为所有子类都必须在生成应用之前完全定义好。

另一方面，黑盒复用具有不同的优点和限制，常常是中型和大型系统中进行对象扩展时更好的选择。在黑盒复用中，对象被通过它指定的接口按照原样使用，客户不能改变对象接口的实现
135 方式。黑盒复用的关键益处在于，通过类似接口定义语言的工具的支持，对象的实现可以独立于它的接口。这可以让开发人员在运行时把接口映射到特定的实现来利用晚期绑定。可以使用静态对象接口来编写客户端，但是随着时间的流逝，可以从支持相同对象接口的更先进的服务中受益。当然，缺点就是支持的服务被限制在那些支持相同接口的服务中。与白盒复用中的接口变化或实现变化相似，对接口的改变通常必须在编译时进行。

白盒复用和黑盒复用的区别反映了面向对象编程（OOP）和面向构件编程（COP）之间的差异。白盒的子类化是面向对象编程的传统特点，而从接口到实现的动态晚期绑定则是面向构件编程的标志。

重新设计软件的结构来减少或消除克隆，要求修改代码来强调对重复软件部分的黑盒复用。在整个生命期中大量使用Cut-and-Paste Programming的软件项目中，恢复你的投资的最有效办法是重构代码集形成强调对功能的黑盒复用的可复用库或构件。如果把这项工作作为一个单独的项目进行，重构过程通常会相当困难、长期而且成本较高。它要求一个强有力的系统架构师来监督和执行这个过程，并协调对软件模块不同扩展版本的优点和限制的讨论。

有效重构以消除多个版本的工作包括3个阶段：代码挖掘、重构和配置管理。代码挖掘是系统性的确定同一段软件的多个版本。重构过程包括开发该代码段的标准版本并把它重新插入到代码集中。配置管理是制定一组策略来帮助预防将来再次出现Cut-and-Paste Programming。大多数时候，除了教育工作，它还需要监督和检测策略（代码评审、审阅和验证）。在所有3个阶段中，管理层的接受对于保证获得资金和支持都是很重要的。

### 5.9.7　示例

136 我们怀疑有一段代码被多个开发机构反复克隆，而且很可能现在还在被克隆。这段代码在数十个开发机构中被看到了几百次。它是一个用来实现链表类的代码文件。它没有使用模板或宏，而是用一个头文件来定义链表中存储的数据结构，因此每个链表被定制成只能操作指定的数据结构。不过，代码的原作者（有传闻说他最初是LISP程序员）在链表代码中引入了一个瑕疵：删除一个条目时链表无法释放它占用的内存，而只是重新编排指针。有些时候，这段代码被修改了来修正这个缺陷；但更经常的是这个缺陷仍然存在。显然代码集是一样的：变量名、指令甚至是格式在每个例子中都是完全一样的，连文件都通常被命名为<prefix>link.c，其中的前缀是一两个字母，隐含地指出该链表管理的数据结构。

### 5.9.8　相关解决方案

Spaghetti Code常包含Cut-and-Paste Programming反模式的多个实例。由于Spaghetti Code的结构并不利于构件复用，很多时候，Cut-and-Paste Programming是复用已有代码段的惟一可用方式。这当然会导致不必要的代码膨胀和维护恶梦。但是经验表明，与使用Cut-and-Paste Programming的实例相比，没有Cut-and-Paste Programming的Spaghetti Code通常是更糟的一团乱麻。

通过实现软件复用过程或组织方式，可以在新的开发中把克隆最小化[Jacobson 1997]。在大型软件开发中，一定程度的克隆是不可避免的。但是，在发生克隆的时候，必须有正规化的过程来把这些克隆体合并到共同的基线[Kane 1997]。

137

#### 小型反模式：Mushroom Management（蘑菇管理）

Mushroom Management也被称为Pseudo-Analysis（伪分析）和Blind Development（盲目开发），常常可以被说明为："让你的开发人员呆在黑暗中，用肥料喂养他们。"一位有经验的系统架构师最近说道："永远不要让软件开发人员和最终用户交谈。"而且，没有最终用户的参与，"风险就是你最后会构建错误的系统。"

**反模式问题**

在某些架构和管理圈子中，有明确的政策要把系统开发人员和最终用户隔离开。需求是通过一些中间人传递的二手信息，这些中间人包括架构师、项目经理或需求分析师。Mushroom Management认为最终用户和软件计划在项目启动时都已经充分理解了需求，而且需求被设想成稳定不变的。

在Mushroom Management中有几个错误的假设：

- 现实中，需求频繁发生变化，会占用大约30%的开发成本。在Mushroom Management项目中，直到项目交付时都没有发现这些变化。用户的接受度总是一个重大的风险，而在Mushroom Management中它成为了至关重要的。
- 最终用户很少能理解需求文档的含义，而在他们体验用户界面原型时更容易把需求的含义可视化。原型可以让最终用户通过与原型特性的比较来明确他们的真实需要。
- 在开发人员没有理解产品的整体需求时，他们不太可能了解需要的构件交互和必要的接口。因此，会做出不良的设计决策，往往导致只有不牢固接口，不能满足功能需求的烟囱构件。

Mushroom Management通过建立一个不确定性环境来影响开发人员。文档化的需求常常不够详细，而且没有有效的途径来获得澄清。为了完成工作，开发人员必须做出假设，从而导致伪分析，也就是在没有最终用户参与的情况下进行面向对象分析。某些Mushroom Management项目完全消除了分析，直接从高层次需求进入到设计和编码。

138

**重构方案**

风险驱动的开发是根据原型和用户反馈进行的螺旋式开发过程。风险驱动的开发是迭代的

增量式开发过程（参阅第7章Analysis Paralysis反模式）的特殊形式。这时，每个增量就是一次外部迭代。也就是说，每个项目增量都包括对用户界面功能的扩展。增量包括用户界面试验，包括实际的体验。试验可以评估每个扩展的接受性和可用性，其结果会在下一次迭代的选择中影响项目的方向。由于项目经常性地评估用户接受性，并使用它作为输入来影响软件的开发，从而可以把用户拒绝接受的风险最小化。

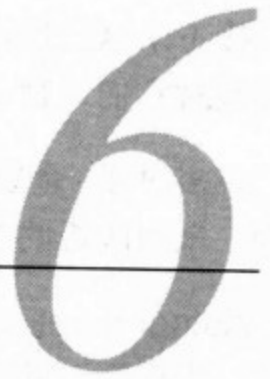
风险驱动的开发最适合于用户界面密集，只需要相对简单的基础结构支持的应用。以本地文件作为存储基础的个人计算机应用程序是风险驱动的开发的最佳对象。

**变化**

在开发团队中包含一位领域专家是在项目决策中包含领域输入的非常有效的方法。无论什么时候出现领域相关的问题，团队成员都有专家在场提供帮助。不过，这种方法的主要风险是该领域专家只代表了来自该领域群体的一个观点。

139

# 第6章 软件架构性反模式



架构性反模式关注系统层和企业层的应用结构及构件结构。虽然软件架构的设计原则相对而言不够成熟，但在软件方面的研究和实际经验都反复证明了架构在软件开发中的极端重要性：

- 良好的架构是系统开发成功的关键因素[Booch 1996，Shaw 1996，Mowbray 1995]。
- 架构驱动的软件开发是构建系统的最有效方法[Booch 1996，Horowitz 1993]。架构驱动的开发优于需求驱动的、文档驱动的和方法驱动的开发。项目往往不管采用哪种方法都可以取得成功，而不是因为某种方法而取得成功[Mowbray 1995]。

软件架构是整体系统架构的子集，而系统架构包括所有的设计和实现方面，包括对硬件和技术的选择。架构的重要原则包括：



- 架构提供整个系统的视图[Hilliard 1996]。这一点将架构与其他关注系统某些部分的分析和设计模型区分开。
- 对整个系统进行建模的有效方法是从多个视角进行建模[ISO 1996]。这些视角分别与系统开发过程中的不同利益相关者和技术专家相关联[Hilliard 1996]。

软件架构与编程在几个方面有所区别。首先，架构师和程序员之间的区别是架构师要考虑他们制定的决策的成本。架构、管理和度量领域的很多人都把复杂性看作是关键的架构设计因素[Shaw 1996]，而软件复杂性和软件成本密切相关[Horowitz 1993]。架构师至少要负责复杂性管理。

软件架构关注软件设计的3个方面[Shaw 1996]：

（1）划分（partitioning）。对软件模块的功能划分。
（2）接口（interface）。模块之间的软件接口。
（3）连接（connection）。用于实现软件模块之间接口连接的技术的选择及其特性。

这些架构决策通常由一组比架构师人数更多的开发人员和维护人员来实现。为了有效管理这些要素，架构师在对人员及其关系进行管理方面要面对重大的挑战。例子包括：与开发人员交流设计、获得管理人员和开发人员的接受，以及管理对设计的实现和扩展。

## 6.1　架构性反模式摘要

接下来这些反模式关注于在建立、实现和管理架构时出现的一些常见问题和错误，本章的剩余篇幅对它们进行了详细的解释。

许多时候，系统之间在设计和技术方面缺乏共同性是导致相关系统间无法提供互用和复用的
142 原因。采用改进的企业架构规划可以协调系统的开发。

**Autogenerated Stovepipe（自生成烟囱）**：在把已有的软件系统迁移到分布式基础结构时会出现这个反模式。当把已有软件接口转换成分布式接口时就会出现Autogenerated StovePipe。如果同样的设计被用于分布式计算，会出现一系列问题。

**Stovepipe Enterprise（烟囱企业）**：Stovepipe System的特点是软件的结构限制了改变系统的能力。重构方案说明了如何对子系统和构件进行抽象来获得改进的系统结构。Stovepipe Enterprise反模式的特点是在一组系统之间缺乏协调和规划。

**Jumble（混乱）**：当横向设计要素和纵向设计要素被混在一起的时候，就会产生不稳定的架构。两类设计要素的混杂限制了架构和系统软件构件的可复用性和稳健性。

**Stovepipe System（烟囱系统）**：子系统被使用多种集成策略和机制以即兴的方式集成到一起，而且都是用点对点的方式集成。每对子系统的集成方法都难以被其他子系统的集成所利用。Stovepipe System反模式是Stovepipe Enterprise反模式的单系统表现，与单个系统内部的子系统如何协调有关。

**Cover Your Assets（隐藏资产）**：由于文档作者想回避做出重要决策，文档驱动的软件开发过程常常会产生没什么用的需求和说明。为了避免犯错误，作者会采用更安全的途径，精心描述各种替代方法。

**Vendor Lock-In（供应商锁定）**：在系统高度依赖于专有架构时就出现了Vendor Lock-In。对架构隔离层的使用可以提供相对供应商特定解决方案的独立性。

**Wolf Ticket（黄牛票）**：如果产品声称具有开放性、符合实际上并没有实施方法的标准，它就是Wolf Ticket。产品交付时采用专有接口，与公布的标准之间可能存在巨大的差别。

**Architecture by Implementation（实现主导架构）**：由于过度自信以及最近开发的系统取得了成功，常常会在后续系统开发中忽视风险管理。针对每个应用系统对通用的架构方法进行裁剪，可以帮助确定独特的需求和风险区域。

143 **Warm Bodies（温暖身体）**：软件项目中的程序员往往在技能和生产率水平上存在很大的差异。这些人中的相当一部分是为了达到人员规模目标而被分配进来的（所以称为“温暖身体”）。有经验的程序员对软件项目的成功是很重要的。所谓的英雄程序员的生产率尤其高，但是只有1/20那么少的人具有这样的天赋。他们产出的可工作软件比普通程序员要高出一个数量级。

**Design by Committee（委员会设计）**：这个经典反模式来自于标准制定组织。Design by Committee会建立过于复杂、缺乏连贯性的架构。对架构角色的澄清和改进的过程推动方

式可以把不良的会议过程重构成具有很高的生产率。

**Swiss Army Knife（瑞士军刀）**：Swiss Army Knife就是过分复杂的类接口。设计者试图提供该类所有可能的用途。为此，他在为了满足所有可能的需要而进行的无用尝试中增加了大量的接口签名。

**Reinvent the Wheel（重新发明轮子）**：软件项目之间普遍缺乏技术转移导致大量的重新发明。可以利用埋藏于遗留资产中的设计知识来减少投入市场的时间、成本和风险。

**The Grand Old Duke of York（约克老公爵）**：平等主义的软件开发过程常常会忽视人的才能对项目的损害。编程技能并不等同于定义抽象的技能。软件开发中似乎有两群不同的人：抽象派和与他们相对的实现派。

144

### 小型反模式：Autogenerated Stovepipe（自生成烟囱）

**反模式问题**

在把已有的软件系统迁移到分布式基础结构时会出现这个反模式。当把已有软件接口转换成分布式接口时就会出现Autogenerated StovePipe。已有软件的设计被用于分布式计算时，会出现一系列问题。例如，已有接口也许要使用细粒度的操作来传递信息，而这些操作在分布式环境中是低效的。先前存在的接口通常是实现所特定的，在用于更大规模的分布式系统中时会导致子系统的相互依赖。本地操作常常会做出关于位置（包括地址空间和本地文件访问）的假设。当在更大规模的分布式系统中暴露出多个已有接口时，会产生过高的复杂性。

**重构方案**

在为已有软件设计分布式接口时，应重新设计接口。应该为分布式接口考虑独立的、粒度较粗的对象模型。多个子系统使用的互用功能应该是新接口的设计中心。通过架构挖掘可以获得独立于特定子系统的设计。考虑到分别编译的软件都要依赖于这些新的设计，新接口的稳定性非常重要。

145

反模式

## 6.2　Stovepipe Enterprise（烟囱企业）

**反模式名称**：Stovepipe Enterprise
**别名**：Islands of Automation（自动化孤岛）
**最常见规模**：企业层
**重构方案名称**：Enterprise Architecture Planning（企业架构规划）
**重构方案类型**：过程
**根源**：匆忙、漠然、思想狭隘
**不平衡的力量**：变化管理、资源管理、技术转移管理
**轶事证据**："我可以拥有自己的（自动化）孤岛吗？"（见图6-1）"我们是独一无二的！"

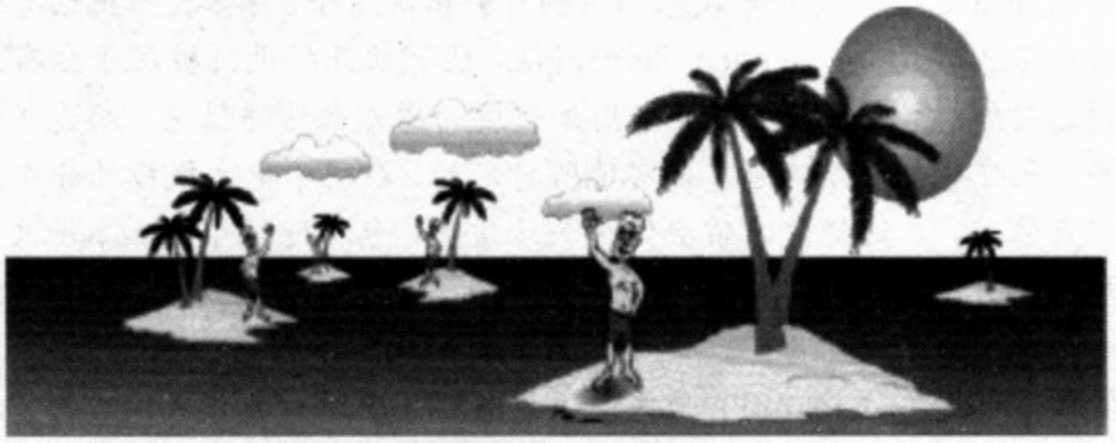


图6-1　自动化孤岛

### 6.2.1　背景

烟囱是用来描述具有即兴架构的软件系统的流行术语。它比喻的是烧木头的炉子，也就是所
147 谓的大肚子炉的排烟管道。由于燃烧的木头会产生腐蚀金属的物质，所以需要经常对烟囱进行维护和修理来避免泄漏。人们常常会用手头持有的任何材料来维修烟管，于是这些烟囱会迅速成为随意维修的大杂烩。因此，使用烟囱这个比喻来描述许多软件系统的即兴生成的结构。

### 6.2.2　一般形式

企业中的多个系统在每个层次上都是独立设计的。缺乏共同性限制了系统之间的互用，阻碍了复用，提高了成本。而且，重新发明的系统架构和服务缺乏具有适应性的高质量结构。

标准和指导原则处于最低的层次。它们的作用就如同跨越企业间多个系统的架构构建法规和分区制法律。往上一个层次是运行环境[Mowbray 1997c]，包括基础结构和对象服务。最上的两层是增值功能服务和任务特定的服务。通过独立地选择和定义所有这些技术，Stovepipe Enterprise
148 建立了与企业的其他部分相隔离的"自动化孤岛"，如图6-2所示。

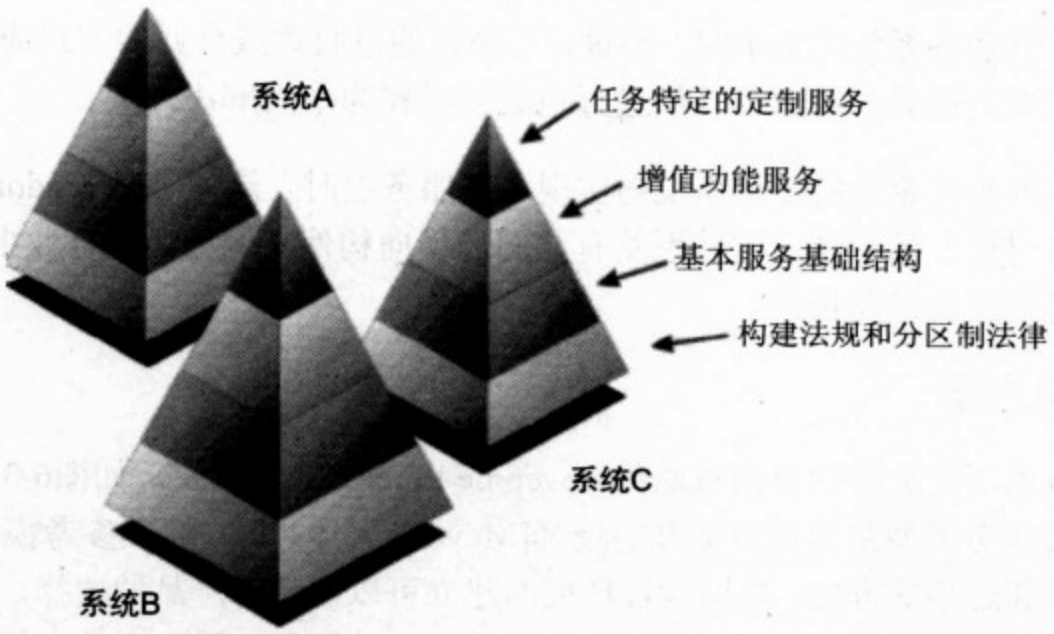



图6-2　Stovepipe Enterprise通常是由每个协调层次上的孤立决策造成

### 6.2.3　症状和后果

- 企业中系统之间不兼容的术语、方法和技术。
- 脆弱、庞大的系统架构和没有文档化的架构。
- 无法扩展系统来支持业务需要。
- 对技术标准的不正确使用。
- 企业的系统之间缺乏软件复用。
- 企业的系统之间缺乏互用性。
- 即使使用了相同的标准，系统也不能互用。
- 变化的业务需求产生过高的维护成本；需要扩展系统才能利用新产品和新技术。
- 雇员更替导致项目中断和维护问题。

### 6.2.4　典型原因

- 缺乏企业技术政策，尤其是：
  - 缺乏标准参考模型[Mowbray 1997a]。
  - 缺乏系统配置文件[Mowbray 1997a]。
- 缺乏对企业开发活动间合作的激励：
  - 竞争的业务领域和主管人员。
- 系统开发项目之间缺乏交流。
- 缺乏关于使用的技术标准的知识。
- 在系统集成方案中没有横向接口。

### 6.2.5　已知例外

对于当今企业层的新系统而言，Stovepipe Enterprise反模式是不可接受的，尤其是在大多数

149 公司都面临扩展业务系统的需求时。不过，当公司通过收购或合并的方式成长时，Stovepipe反模式很可能会发生。这时，对部分系统进行包装可以作为中间解决方案。

另一个例外是在企业的多个系统内实现通用服务层时。这通常是Vendor Lock-In反模式（在本章后续部分讨论）的表现。这些系统有共同的横向构件，例如在银行业中，诸如DB2和Oracle的数据库就是这样的横向构件。

### 6.2.6　重构方案

在多个技术层次上进行协调对避免Stovepipe Enterprise很重要，如图6-3所示。首先，可以通过为标准定义参考模型来协调对标准的选择[Mowbray 1997a]。标准参考模型为企业中的系统定义了共同标准和迁移的方向。共同运行环境的建立可以协调对产品的选择，控制对产品版本的配置。定义系统配置文件来协调对产品和标准的使用，对保证获得标准带来的效益、复用和互用性非常关键。至少应该有一个系统配置文件定义了跨系统的使用协定。

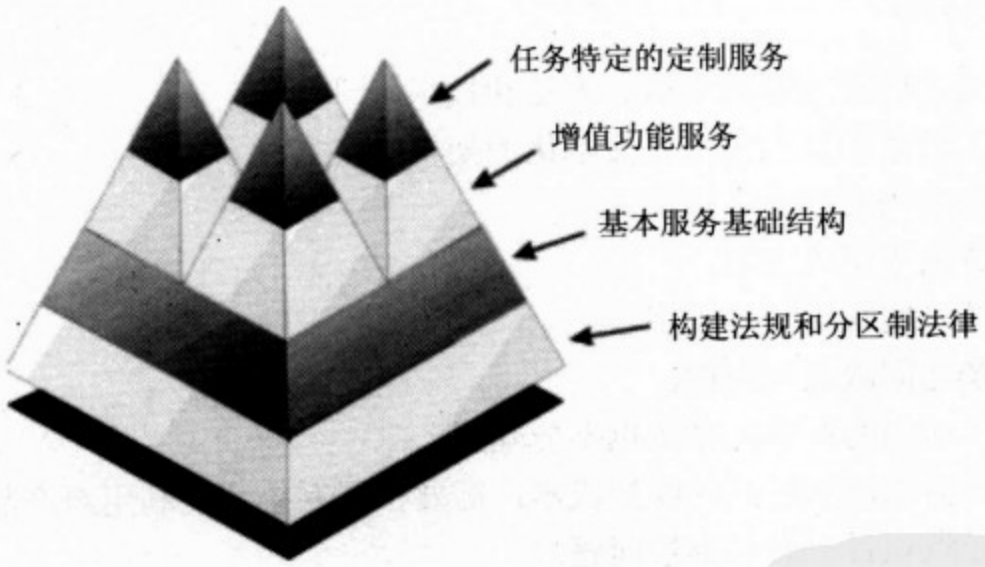



图6-3　经过协调的技术让共同基础结构和标准的建立成为可能

大企业根据大量的经验提出了一些有用的定义面向对象架构的协定，它们可以应用于很多的
150 机构。大规模架构要面对的一个关键挑战是在考虑技术政策和需求的条件下定义详细的跨系统互用协定。对极大规模的企业，经验显示为了正确确定互用性挑战的范围并解决它们，需要4个需求模型和4个规范模型，如图6-4所示。

需求模型包括：

（1）开放式系统参考模型。
（2）技术配置文件。
（3）运行环境。
（4）系统需求配置文件。

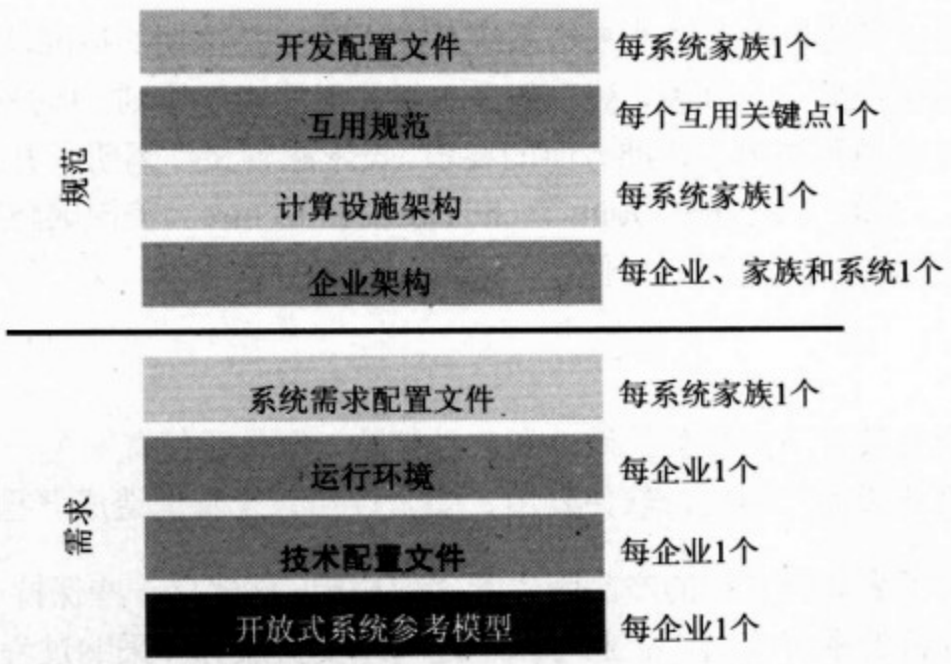



图6-4　多层次协调：技术政策、需求和规范在大企业中可能是必需的

规范模型包括：

（1）企业架构。
（2）计算设施架构。
（3）互用规范。
（4）开发配置文件。



接下来的部分说明了这些模型，它们都是企业整体架构规划的一部分。该规划在项目间提供了有效的协调，减少了对点对点互用解决方案的需要。

**开放式系统参考模型**

开放式系统参考模型包含用于系统开发项目的高层次架构图和目标标准列表。该模型的目的是确定项目的所有候选标准，协调开放式系统策略。

IEEE POSIX 1003.0标准是一个现成的此类参考模型。POSIX（Portable Operating System Interface，可移植操作系统接口）的这一部分列出了相当广泛的开放式系统标准，包括它们的适用性、成熟度、商业支持和其他要素。该POSIX标准是很多企业特定的参考模型的起始点。

**技术配置文件**

在多年前开放式系统参考模型刚提出时，它们被看作对系统互用的完整解答。不过，开发人员并不确定这些模型会如何影响它们的项目。关键问题在于参考模型定义了未来的架构目标，但是没有说明实现它的时间框架。而且，由于标准制定组织的活动，每年都有大约1/3的内容会改变。

于是，人们提出了技术配置文件来给系统开发人员定义短期的标准规划。技术配置文件是从参考模型提取出的标准的简明列表，被看作是一组灵活的指导原则。但是技术配置文件常常对当前的和新的系统开发项目提出了标准方面的要求。技术配置文件阐明了开发人员要就参考模型中的标准做些什么；例如，US-DOD Joint Technical Architecture（美国国防部联合技术体系）就是为当前的实现确定标准的技术配置文件。

### 运行环境

大多数大型企业具有不同种类的硬件和软件架构，但是即使有一个一致的基础结构，不同的
152 安装实践也会对企业级的互用性、软件复用、安全性和系统管理造成严重的问题。

运行环境定义了企业所支持的产品版本和安装协定，并建立一些保持了局部灵活性的指导原则来支持研发和独特的系统需求。企业可以通过技术支持服务和采购过程来鼓励遵守这些协定。也就是说，企业可以通过让建议的环境成为最容易获得、最容易支持的系统配置来影响到对它们的采用。对运行环境的改变则必须在局部用额外的成本来支持。

### 系统需求配置文件

企业架构规划常常会产生范围宽广的高层次需求文档。对任何特定的系统家族而言，由于这些信息的庞大数量，需求对开发人员可能都是不够清晰的。系统需求配置文件是对一组相关系统构成的家族的关键需求的摘要。它的时间范围是短期的。理想条件下，该文档的长度只有几十页，澄清了构件系统和应用开发项目意图实现的目标。

系统需求配置文件确定了系统能力的必要范围，因此是对企业需求规划的测量点。对企业规划模型的衡量是通过架构和设计规范（后续部分将介绍）来进行的，它通过面向对象模型表示出来，包括一组面向对象的软件架构。

### 企业架构

企业架构是一组图示和表格，从不同利益相关者的角度来定义系统或系统家族。因此，企业架构包括整个系统的不同视图。它描述了当前的和将来的时间范围，每个视图解决不同的主要利益相关者，包括最终用户、开发人员、系统操作员和技术专家提出的问题。

由于企业架构让跨项目的技术交流成为可能，在不同项目间保持架构视图和注记符号的一致性就很重要。当项目具有共同的架构时，就可以发现复用和互用的机会。由于项目个体对技术细
153 节有最充分的了解，可以汇总项目特定的架构来得到准确的、企业范围的架构。

### 计算设施架构

和刚刚解释的一样，企业架构是最终用户和架构师的重要交流工具。剩下的规范则详细说明了定义互用性和复用接口的计算架构。

计算设施架构（CFA）确定和定义系统家族中的互用性关键点。每个设施确定一组在互用规范中详细定义的API和公共数据对象。CFA把企业的互用性要求划分成可管理的规范；它还为这些设施定义了由优先级和进度表构成的路线图。这对启动和引导互用规范的制定过程是必需的。

在CFA中应包含哪些设施上取得一致对很多企业来说都是很关键的挑战。在这些设施所扮演的与外部需求有关的角色、对系统独立性的要求、共同抽象的定义和限制设施范围的必要性等方面，存在着大量的错误理解。

**互用规范**

互用规范定义了计算设施的技术细节。典型的互用规范包括使用IDL定义的API和公共数据对象定义。

互用规范以独立于任何特定系统的子系统实现的形式建立了互用性的关键点。架构挖掘是建立这种规范时尤其有效的过程[Mowbray 1997c]。在系统维护时，互用关键点会成为系统扩展的增值入口点。

**开发配置文件**

仅仅只有互用规范还不足以保证成功的集成，因为不同的实现者可能对API的语义做出不同的解释。稳健的API设计中具有内在的灵活性来允许扩展和复用，而它们的用途的细节往往在开发过程中才被发现。这种细节中的一部分也许是独特地用于特定系统集的。

开发配置文件记录了对保证互用性和成功集成必不可少的实现计划和开发人员间的协议。开 154
发配置文件确定使用哪部分API规范、对规范的局部扩展和用于协调集成的协定。

对所有这些模型都进行配置控制很重要，而且开发配置文件是工作用的文档，会在整个开发和维护生命周期中发展。可能对单个API规范存在多个开发配置文件，每个分别负责解决特定领域或特定系统家族的集成需要。

### 6.2.7　示例

图6-5中的系统1和系统2代表同一个企业中的两个Stovepipe System。虽然这两个系统在很多方面很相似，但是它们缺乏共同性。它们使用不同的数据库产品和不同的办公自动化工具，具有不同的软件接口，使用不同的图形用户界面。这两个系统间潜在的共同性没有被认识到，所以设计人员和开发人员没有对此加以利用。

要解决该反模式，企业首先定义了标准参考模型。图6-6显示的该模型选择了一些基线标准用于跨越所有系统的信息交换。接下来的步骤是为运行环境选择产品。在这个例子中，两种数据 155
库产品都被选中，但只选择了一个办公自动化工具。这是该企业受到支持的未来迁移方向。企业可以通过企业产品许可、培训和技术支持来推广这个运行环境。在这个层次还定义了配置文件，用于在可复用服务实现中利用这些技术和共同接口。GUI应用构成了剩下的系统特定的实现。

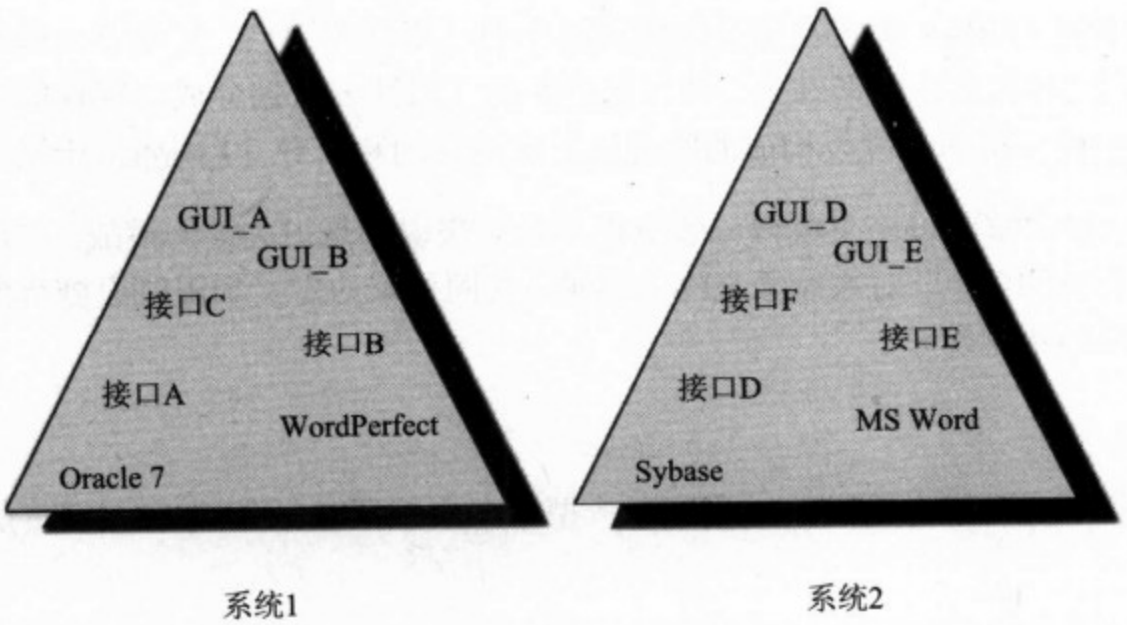



图6-5　Stovepipe Enterprise中的两个系统

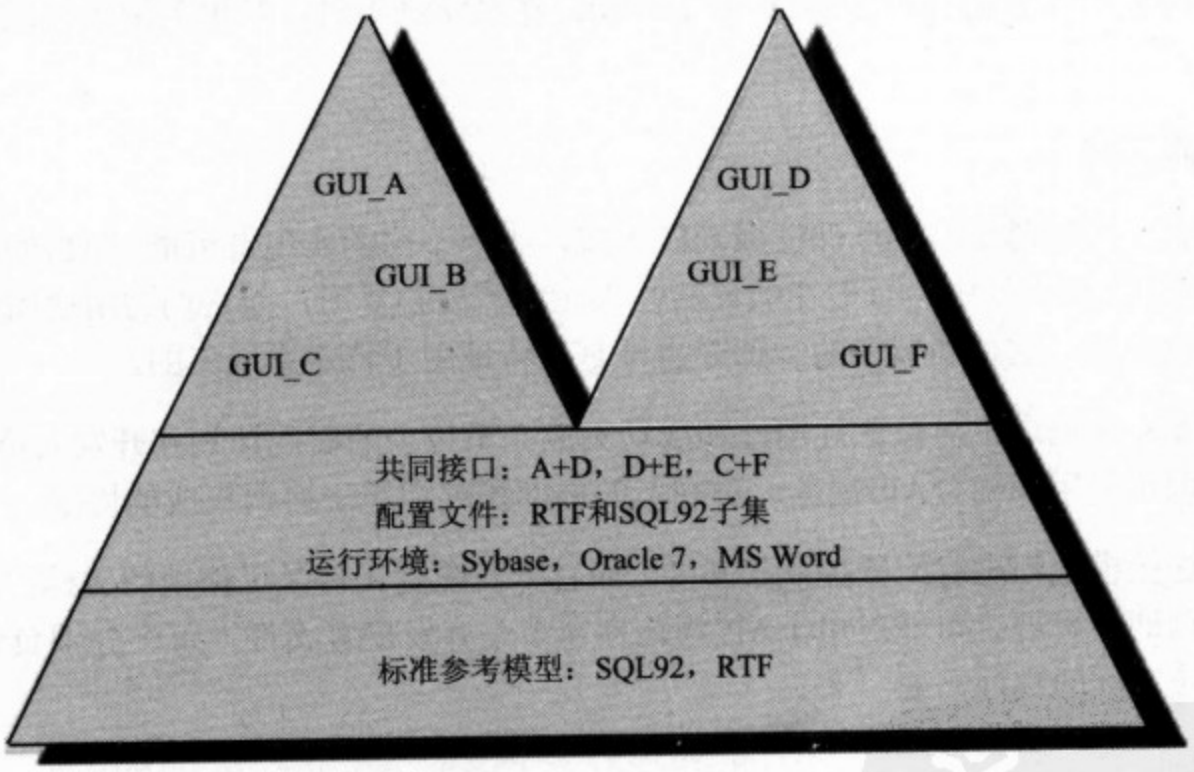



图6-6　重构方案：技术协调

### 6.2.8　相关解决方案

Reinvent the Wheel反模式是Stovepipe System问题的一个子集，关注的是由于开发项目之间缺乏交流导致的缺乏成熟的设计和实现。

标准参考模型、运行环境和配置文件是来自于*CORBA Design Patterns* [Mowbray 1997a]一书的解决方案。它们都是Stovepipe Enterprise反模式解决方案的重要组成部分。

标准参考模型的例子包括IEEE POSIX.0、美国国家标准与技术研究院的Application Portability Profile（APP）以及美国国防部信息管理技术体系框架（TAFIM）第7卷。在www-ismc.itsi.

disa.mil/ciiwg/ciiwg.html可以找到通用的接口和配置文件。 156

### 6.2.9　对其他视角和规模的适用性

Stovepipe Enterprise常常是管理形成的机构组织边界造成的后果。抑制了交流和技术转移的组织结构会在机构内部产生隔离，导致Stovepipe Enterprise标志性的协调缺乏。Stovepipe Enterprise反模式对管理的影响在于每个系统的开发都要涉及大量不必要的风险和成本。由于系统不互用而且难以集成，整体组织效能就受到了影响。Stovepipe Enterprise严重削弱了开发机构承受业务需求变化的能力。企业正在逐渐面对一种被称为敏捷系统（agile system）的需求，它们可以适应业务处理过程中的变化，因为它们早已支持跨越企业中大部分或全部系统的互用。

开发人员也会受Stovepipe Enterprise的影响，因为他们常被要求建立脆弱的解决方案来连接单独架构出的系统。这些接口难以维护和复用，而且缺乏技术协调导致建立这些接口的工作相当困难。 157
中间件解决方案和商业产品（数据库引擎）的组合往往需要建立这样的联系才能获得互用性。

**小型反模式：Jumble（混乱）**

**反模式问题**

当横向设计要素和纵向设计要素被混在一起的时候，就会产生不稳定的架构。纵向设计要素依赖于单个应用和特定软件实现。而横向设计要素在多个应用和特定实现之间是共同的。默认情况下，开发人员和架构师会把两类要素混合到一起。但是这样就限制了架构和系统软件构件的可复用性和稳健性。纵向要素导致的软件依赖性限制了扩展和复用。两类设计要素的混杂降低了架构和系统软件构件的可复用性和稳健性。

**重构方案**

第一步是确定横向设计要素，把它们划分到隔离的架构层中。然后使用横向要素来捕捉架构中共同的互用性功能。例如，横向要素是对特定子系统实现的抽象：

（1）增加纵向要素作为扩展来获得专用功能和提高性能。

（2）在架构中使用元数据。

（3）用动态要素（元数据）替代设计中的静态要素（横向和纵向的）。

架构中横向、纵向和元数据要素的适当平衡可以引向结构良好、可扩展、可复用的软件。

**背景**

要花一些时间才能完全理解横向和纵向设计要素的含义。在本书的姊妹篇*CORBA Design Patterns*[Mowbray 1997c]中对这些主题进行了更深入的探讨。特别是横向－纵向－元数据（HVM）模式和相关的CORBA设计模式建立了软件架构设计的关键原则。Jumble反模式说明了对这些原则最常见的误用。 158

反模式

## 6.3　Stovepipe System（烟囱系统）

**反模式名称**：Stovepipe System
**别名**：Legacy System（遗留系统）、Uncle Sam Special、Ad Hoc Integration（即兴集成）
**最常见规模**：系统层
**重构方案名称**：Architecture Framework（架构框架）
**重构方案类型**：软件
**根源**：匆忙、贪婪、物质、懒惰
**不平衡的力量**：复杂性管理、变化管理
**轶事证据**："这个软件项目要超支了；它反复地跟不上进度要求；我的用户还没有得到期望的功能；而且我没法修改系统。每个构件都是一个烟囱。"

### 6.3.1　背景

Stovepipe System是广泛用于指代具有不受欢迎品质的遗留软件的贬义名称。在该反模式中，我们把这些负面品质的原因归结于系统的内部结构。改进的系统结构可以让遗留系统得到发展，满足新的业务需要，无缝地结合新的技术。通过应用我们建议的解决方案，系统可以获得Stovepipe System所不具备的全新适应能力。

### 6.3.2　一般形式

159 Stovepipe Enterprise是在一组系统间缺乏协调和规划，而Stovepipe System反模式就是它的单系统形式。Stovepipe System反模式解决的是如何协调单个系统内部的子系统的问题。该反模式中的关键问题是缺乏通用的子系统抽象，而在Stovepipe Enterprise中，关键问题是缺乏通用的多系统协定。

子系统被使用多种集成策略和机制以随意的方式集成到一起。如图6-7所示，所有子系统都是以点对点的方式集成，每对子系统的集成方法都不容易被其他子系统的集成所利用。而且，由于有很多隐含的对系统配置、安装细节和系统状态的依赖，系统实现很脆弱。系统难以扩展，而且扩展会增加额外的点对点集成链接。在Stovepipe System的整个生命周期中，每集成一种新的能力和变化，都会增加系统的复杂性。于是，系统的扩展和维护越发难以处理。

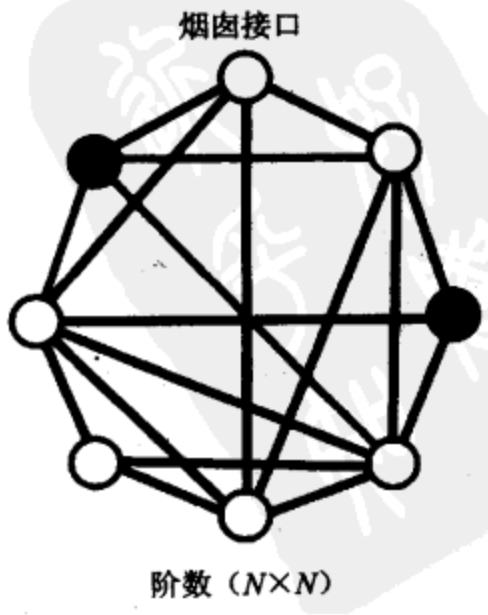



图6-7　烟囱接口要求点对点的集成，导致阶数为（N×N）的系统成本和复杂性

### 6.3.3　症状和后果

- 架构文档和实现的软件之间存在巨大的语义差距；文档与系统实现不对应。
- 架构师不熟悉集成方案的关键方面。
- 项目在没有明显原因的情况下超支和错过进度安排。
- 需求变化的实现成本很高，系统维护会产生非常高的成本。
- 系统可能符合大部分纸面上的需求，但是不能满足用户的期望。 160
- 用户必须发明一些方法来绕过系统的限制。
- 要遵循复杂的系统和客户端安装过程，无法进行自动化。
- 不可能与其他系统互用，也无法支持集成的系统管理和系统间安全功能。
- 改变系统的难度日益增加。
- 对系统的修改越来越容易产生新的严重错误。

### 6.3.4　典型原因

- 使用多种架构机制来集成子系统；缺乏共同机制导致难以说明和修改架构。
- 缺乏抽象；每个子系统的接口都是独一无二的。
- 对元数据的使用不足；没有可用的元数据来支持在不改变软件的情况下对系统进行扩展和重配置。
- 实现的类之间采用紧耦合，要求过多的服务特定的客户端代码。
- 缺乏架构前景。

### 6.3.5　已知例外

软件产品的研发活动常常会使用Stovepipe System反模式来迅速得到一个解决方案。对原型和模型来说，这是完全可以接受的。有些时候，由于缺乏领域知识，可能会要求先开发一个Stovepipe System来获取领域信息，以便用于构建更稳健的系统，或者用于把初始的系统发展成改进的版本[Foote 1997]。选择使用某个供应商的产品而不是重新发明轮子，也有可能会导致Stovepipe System反模式和Vendor Lock-In反模式。

### 6.3.6　重构方案

对Stovepipe System反模式的重构方案是一个构件架构，它可以支持对软件模块的灵活替换。 161
对子系统进行抽象的建模，因此暴露出的接口数量比子系统实现的数量要少很多。对软件模块的替换可以是静态的（编译时构件替换）也可以是动态的（运行时动态绑定）。定义构件接口的关键是发现适当的抽象。子系统抽象对系统的互用要求进行建模，而没有暴露子系统之间和实现特定的细节之间不必要的差异，如图6-8所示。

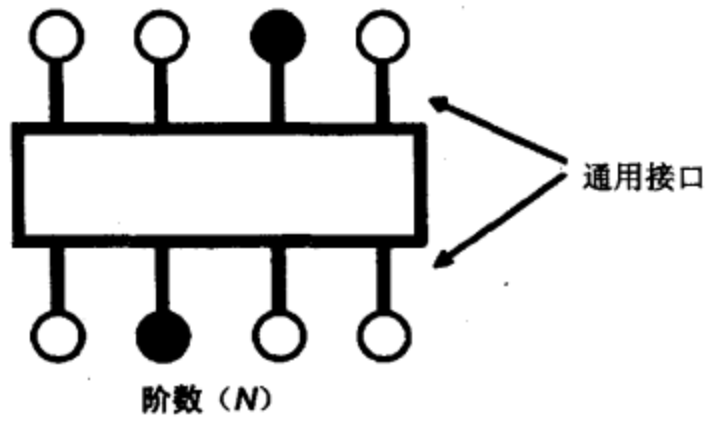



图6-8　构件架构使用通用接口来对互用需要进行建模

要定义构件架构，应该选择应用的主体要支持的基础功能层次。一般而言，该层次应该较低，关注互用性的单个方面，例如数据交换或转换。然后定义一组支持该基础功能层次的系统接口；我们推荐使用ISO IDL来进行定义。大部分服务都具有额外的接口来表达更细粒度的功能要求，因此构件接口应该足够小。

具有基础层次的构件服务让同一个领域中的所有客户端使用，可以鼓励开发瘦客户端，它们能够很好地与现有及未来的服务一起工作而不用进行修改。瘦客户端指的是不需要详细了解服务和系统架构的客户端；可以利用框架来支持和简化它们对复杂服务的访问。具有多个支持插入的实现可以提高客户端的稳健性，因为可以有多种可选方法来满足它们的服务要求。

应用将会有根据更专门的（纵向）接口编写的客户端。纵向客户端应该保持不受新构件接口
162 增加的影响。可以按照横向接口来编写只需要基础层次功能的客户端，它们应该更稳定，易于被新应用或其他已有应用支持。横向接口应该通过抽象只提供构件在基础层次的功能，而隐藏所有更低层的细节。应该编写客户端来处理接口所需要的所有数据类型，以便支持未来在横向构件实现间的交换。例如，如果返回了一个“任意”类型，客户端就应该能够处理“任意”所能够代表的所有类型。无可否认，对于不支持在运行时传递新用户定义类型的CORBA实现，可能必须在横向层次上进行类型管理；尤其是可能必须把纵向类型转换成编译时已知的横向类型。

在构件架构中结合元数据对服务发现和服务区分非常关键。通过命名和交易服务可以提供基本水平的元数据支持[Mowbray 1997c]。命名服务让发现已知对象成为可能；而交易服务列出了可用的服务和它们的属性供客户端来发现。可互用的命名服务被扩展以结合一些交易能力。要更好地将客户端和服务解耦，往往需要更大量地使用元数据。例如，数据库服务的模式元数据可以帮助客户端适应不同的模式和对模式的改变[Mowbray 1995]。

### 6.3.7　示例

图6-9展示了一个典型的Stovepipe System。有3个客户端子系统和6个服务子系统。每个子系统具有一个独特的软件接口，而且每个子系统实例都被建模成类图中的一个类。建立系统时，每个客户端都有独特的接口软件来对应每个集成的子系统。如果增加或替换了另外的子系统，就必

须用额外的代码修改客户端来集成新的独特接口。

该例子的重构方案考虑了子系统间的共同抽象（见图6-10）。由于每类服务各有2个，每个模型就可能具有一个或更多共同的服务接口。然后可以包装每个特定设备或服务来支持共同的接口抽象。如果给系统增加来自这些抽象子系统类别的额外设备，就可以把它们透明地集成到已有的系统软件。添加的交易服务增加了发现和区分抽象服务的能力。 163

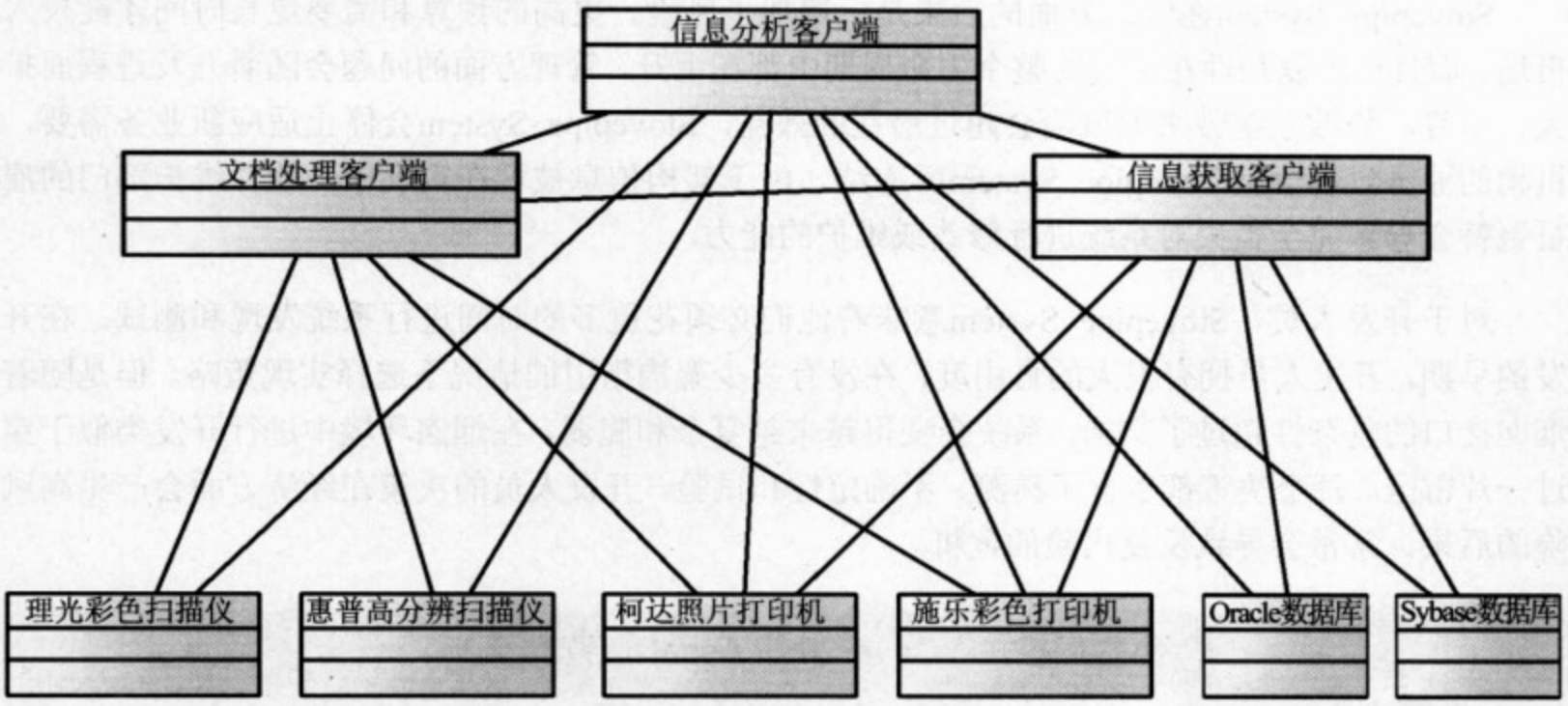



图6-9　客户端/服务器Stovepipe System

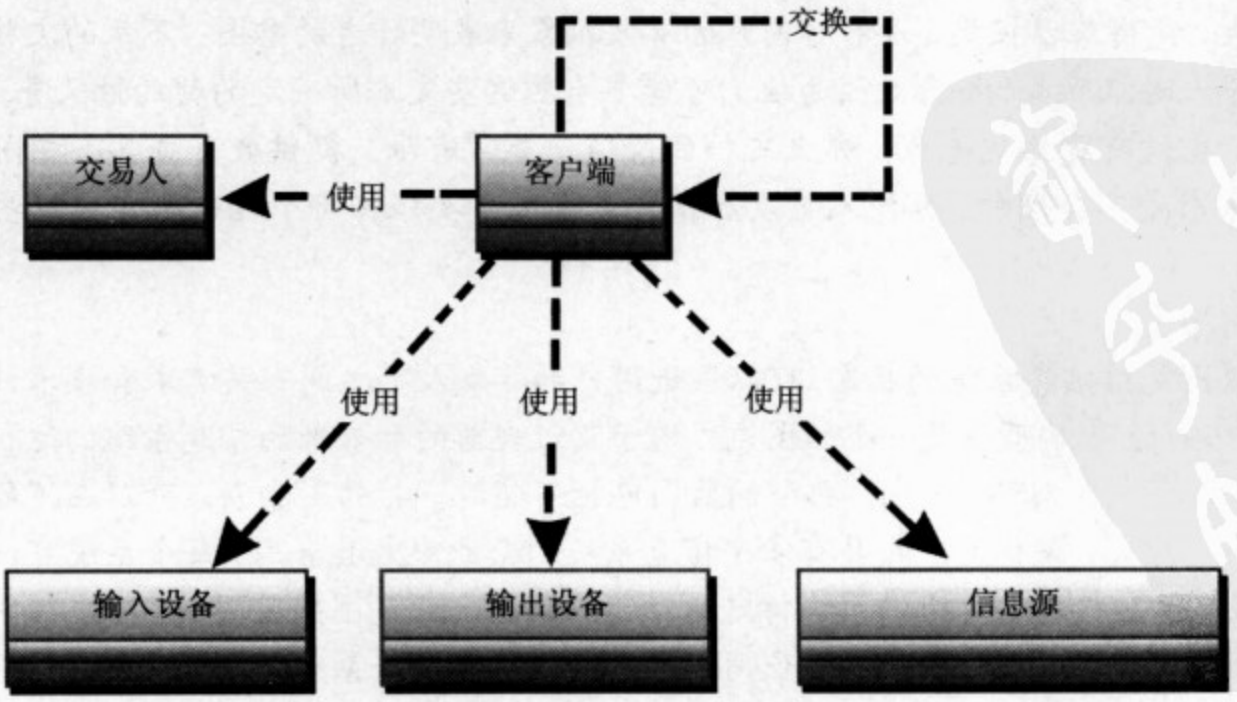



图6-10　重构的架构对子系统间的差异进行了抽象

### 6.3.8　相关解决方案

Stovepipe Enterprise反模式描述了烟囱实践如何在企业规模上进行传播。请注意Stovepipe
164 Enterprise针对多系统的问题，涉及比单个系统更大规模的架构。

### 6.3.9　对其他视角和规模的适用性

Stovepipe System在管理方面的后果是：增加了风险、更高的预算和需要更长时间才能投入市场。而且由于复杂性在系统的整个生命周期中都在上升，管理方面的问题会随着开发进程而扩大。最终，修改系统带来的风险会超过潜在的效益，Stovepipe System会停止适应新业务需要，机构的业务过程会被Stovepipe System所冻结。由于架构信息被埋在实现中，软件维护部门的雇员更替会导致完全丧失对系统进行修改或维护的能力。

对于开发人员，Stovepipe System意味着他们必须花更多的时间进行系统发现和测试。在开发的早期，开发人员拥有很大的自由度，在没有多少架构指引的情况下选择实现策略。但是随着烟囱接口的复杂性超过了文档，系统会变得越来越复杂和脆弱。在烟囱环境中进行开发类似于穿过一片雷区。每个决策都包含了猜测、不确定性和试验。开发人员的决策在经济方面会产生高风
165 险的后果，常常会导致反复出险的危机。

#### 小型反模式：Cover Your Assets（隐藏资产）

**反模式问题**

由于文档作者想回避做出重要决策，文档驱动的软件开发过程常常会产生不太有用的需求和说明。为了避免犯错误，作者会采用更安全的途径，精心描述各种替代方法。结果生成的文档体积庞大，变得难以捉摸。没有对内容的有效抽象来表明作者的意图。不幸的文档读者必须仔细研读令人思维麻木的细节，因为他们可能要承担这些文本所确定的契约性义务。如果没有做出决策，也没有建立优先级，那么文档的价值就非常有限。提供数百页具有相同重要性或强制性的需求是不合理的。开发人员没有获得多少关于要根据哪个优先级实现哪些内容的有益指引。

**重构方案**

架构蓝图是对信息系统的抽象，可以促进用户和开发人员之间有关需求和技术计划的交流[Blueprint 1997]。架构蓝图是一小组图表，用于交流当前的和未来的信息系统的运行架构、技术架构和系统架构[C4ISR 1996]。典型的蓝图包括不超过一打的图和表，可以在不超过一个小时的时间内介绍完。架构蓝图在具有多个信息系统的企业中尤其有用。每个系统可以建立自己的架构蓝图，然后机构可以根据系统特定的细节汇编出企业范围的蓝图。蓝图应该同时体现已有的系统和规划的扩展的特点。可以使用扩展来协调跨越多个系统的架构规划。由于架构蓝图
166 允许多个项目描绘出它们所使用的技术，互用和复用的机会都得到了增强。

反模式

## 6.4　Vendor Lock-In（供应商锁定）

**反模式名称**：Vendor Lock-In

**别名**：Product-Dependent Architecture（依赖产品的架构），Bondage and Submission（束缚和屈服），Connector Conspiracy（连接器阴谋）

**最常见规模**：系统层

**重构方案名称**：Isolation Layer（隔离层）

**重构方案类型**：软件

**根源**：懒惰、漠然、傲慢/无知（轻信）

**不平衡的力量**：技术转移管理、变化管理

**轶事证据**：我们常会遇到软件项目声称它们的架构是根据特定供应商或特定产品线构建的。

其他的轶事证据在产品升级和安装新应用时会出现："当我试图把新数据文件读取到旧版的应用中时，它让我的系统崩溃了。""只要你把数据读入到新应用中，就再也不能把它取出来。""旧软件的行为就像感染了病毒一样，不过也很可能只是因为新应用数据造成的。""我们的架构是……再问一下我们的数据库的名字是什么？"

> "建在坚固岩石上的是丑陋的房子：来看看我建在沙子上的华丽宫殿！"
>
> ——Edna St.Vincent Millay（美国女诗人）

### 6.4.1　背景

这个反模式中最坏的情况是，你的数据和软件的许可完全分配给了在线服务，而某一天跳出了如图6-11所示的模式对话框。 167

交互式字处理比语言格式化技术（比如SGML）更受欢迎，因为它可以让用户在计算机屏幕上看到最终格式，并打印出和屏幕显示完全一样的副本。这种能力被称为"所见即所得"（WYSIWYG）。Vendor Lock-In的一种普遍变形是被称为"所见类似于所得"（WYSISLWYG，发音是"weasel wig"）的现象。最近，由于微软在桌面系统所占据的统治地位，产品功能失常导致文档打印出的版本和它们在屏幕上的显示差异显著。例如，绘图中的符号会发生变化或消失，嵌入的对象常常会被打印成命令字符串（类似于"{EMBEDDED POWERPOINT FIGURE}"）。来自不同版本的同一个微软产品的文档会导致公司网络上的支持问题和系统的崩溃。许多公司不鼓励或禁止混合使用产品的不同版本。由于在文档交换上对微软产品的机构性依赖，很难避免这种形式的Vendor Lock-In和它的产品功能失效。 168


图6-11　在Vendor Lock-In下你将来可能遇到的情况

### 6.4.2　一般形式

软件项目采用某项产品技术，变得完全依赖于该供应商的实现。在进行升级时，软件会改变从而出现互用问题，所以要求持续的维护来保证系统运行。此外，期待的新产品特性常常会延误，造成进度延期，无法完成期望的应用软件功能。

### 6.4.3　症状和后果

- 应用软件维护周期受商业产品升级驱动。
- 供应商承诺的产品特性被延误或从未交付，于是导致无法交付应用的升级。
- 与宣传的开放式系统标准相比，产品变化显著。
- 如果完全错过某个产品升级，往往需要重新采购和重新集成产品。

### 6.4.4　典型原因

- 由于没有有效的过程来保证遵守标准，产品不同于公布的开放式系统标准。
- 完全根据市场和销售信息，而不是根据更详细的技术评审来选择产品。
- 没有把应用软件和对产品的直接依赖隔离开的技术方法。

169

- 应用编程需要更深入的产品知识。
- 产品技术的复杂性和一般性严重超过了应用的需要，对产品的直接依赖导致无法管理应用系统架构的复杂性。

### 6.4.5　已知例外

在单个供应商的代码构成应用所需代码的主体时，Vendor Lock-In反模式是可以接受的。

### 6.4.6　重构方案

对Vendor Lock-In反模式的重构方案被称为隔离层。如图6-12所示，隔离层可以分隔软件包和技术。可以使用它从位于下层的中间件和平台特定的接口之上提供软件可移植性。在符合下列的一个或多个条件时，可以应用该解决方案：

- 应用软件和更低层基础结构的隔离。这里的基础结构可能包括中间件、操作系统、安全机制或其他低层机制。
- 在受影响的软件的生命周期中，预期下层基础结构会发生改变。例如，新产品发布或计划迁移到新基础结构。 170
- 更方便的编程接口是有益的或必需的。对意图实现的应用或系统而言，基础结构提供的抽象水平过于简单或过于灵活。
- 需要跨越很多系统一致地处理基础结构。必须制定对基础结构接口进行默认处理的某些重量级协定。
- 在生命周期中必须支持多个基础结构，或者需要同时支持它们。

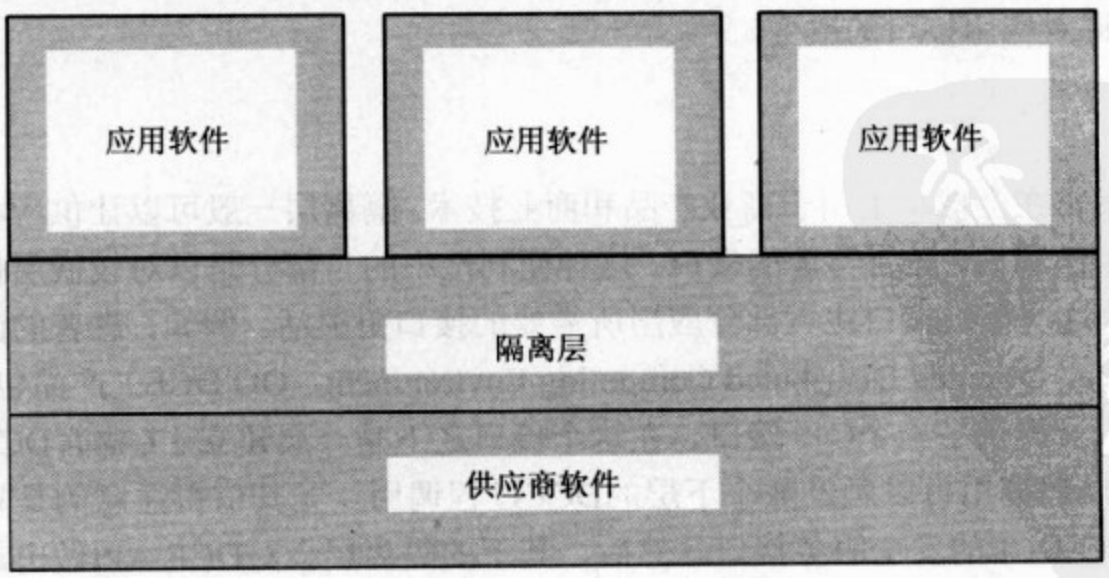



图6-12　隔离层把应用软件和依赖于产品、可能改变的接口隔离开

解决方案要求建立一层软件来对内在的基础结构或依赖于产品的软件接口进行抽象。该层提供了一个应用接口，把应用软件与下层接口完全隔离开。该应用接口应该为需要的能力实现一个便于使用的、语言特定的接口。隔离层软件应保证对某些基础结构调用和参数提供默认处理，但

也会适当地暴露其他的细节。

应跨越多个系统开发项目使用该隔离层来保证互用性、一致性和隔离性。要实现它，可以根据需要把隔离迁移到新基础结构；而且在基础结构更新时更新隔离层。在所有情况下，不管基础结构如何变化，都要维持相同的应用软件接口。

在必须同时支持的多个基础结构间安装网关[Mowbray 1997c]，以及在基础结构迁移时安装正向和逆向网关[Brodie 1995]也是必需的。

这一解决方案带来的益处有：

- 降低基础结构迁移的风险和成本。
- 排除由于基础结构变化造成的过时。
- 减少由于基础结构变化而要求的软件升级的风险和成本。
- 保证给大多数应用程序员提供劳动强度更低的、更廉价的编程接口。
- 支持透明地同时使用多个基础结构。
- 强制实施对灵活的接口和参数进行协调的默认处理。
- 把基础结构知识和应用知识隔离开，从而允许由一小队基础结构开发人员来维护隔离层，而同时程序员的主体可以使用隔离层软件的定制接口。

171

该方案的其他效果包括：

- 潜在地必须在多个平台和基础结构上迁移和维护隔离层。
- 必须协调负责定义初始隔离层接口的开发人员。
- 必须协调对应用接口的改变。

### 6.4.7　变化

该解决方案常在全球层上用于商业产品和商业技术。隔离层一般可以让供应商为某种较低层的技术提供一个便利的、语言特定的接口。这种便利性中的一部分是以对较低层接口的默认处理的形式出现的，这些低层接口比大部分应用所需要的接口更灵活。例如，惠普的面向对象分布式计算环境（Object-Oriented Distributed Computing Environment，OO DCE）产品包括一个隔离层，并为应用开发人员提供了一个C++接口。在这个接口之下是一层建立于C语言DCE环境的隔离层软件。对C++的API调用可以触发数个下层的DCE过程调用。尤其值得注意的是只需要两个调用就可以初始化OO DCE的安全服务接口。然后，其下的隔离层会对DCE API做出超过50个调用来使用遗留DCE安全服务完成这个初始化。

隔离层方案最适用于企业层。不过，个别系统也采用这个方案来提供中间件隔离。例如，Paragon Electronic Light Table（ELT）产品在被称为ToolTalk的Common Desktop Environment（CDE）中间件基础结构上使用了一个隔离层。通过隔离ToolTalk，Paragon可以很容易地把它的产品迁移

到CORBA基础结构，同时支持CORBA和ToolTalk基础结构。

### 6.4.8　示例

下面的例子是3个已知的对隔离层解决方案的使用。

（1）基于HP ORB增强版的ORBlite框架把应用软件与多种语言映射和网络协议隔离开[Moore 1997]。这样，由于OMG映射在采纳和修订过程中的发展，ORBlite可以支持对C++语言的多种语言映射[Moore 1997]。

172

（2）虽然OpenDoc不再在它的创建者的产品战略中占有一席之地，但它使用了一些令人感兴趣的技术方法，包括隔离层解决方案。OpenDoc Parts Framework（OPF）使用ISO IDL来定义，为OpenDoc的复合文档接口建立了更高层次的C++编程接口。OPF包括到操作系统函数（包括图形显示）以及OpenDoc函数的接口。这样，OPF为中间件、窗口显示和操作系统提供了源代码可移植接口。使用OPF编写的复合文档处理部分可以通过重编译和链接移植到OS/2、MacOS和Windows95。来自负责OpenDoc的团体Component Integration Labs的名为LiveObjects的测试能力保证了构件的可移植性和互用性。

（3）EOSDIS（Earth Observing System Data and Information System，地球观测系统数据和信息系统）是由NASA提供资金支持的大规模信息获取项目。EOSDIS中间件抽象层被用于隔离应用软件和发展中的中间件。EOSDIS的初始原型使用了beta测试版的CORBA产品。这些原型构建工作被证实是不成功的，主要是因为使用该beta测试版产品时遇到的困难。虽然程序管理层承认未来还需要对CORBA的支持，但是仍然选择了专有的面向对象的DCE扩展用于短期实现。管理层也不想完全依赖于专用接口。通过增加一个中间件抽象层来给EOSDIS应用软件屏蔽对中间件的选择，这个问题得以解决。该层隐藏了对象创建、对象激活和对象调用中的差异。

### 6.4.9　相关解决方案

该模式与Object Wrapper模式有关[Mowbray 1997c]。Object Wrapper模式提供了单个应用与单个对象基础结构间的隔离。隔离层模式提供了多个应用与多个基础结构之间的隔离。这个反模式还与Profile模式有关[Mowbray 1997c]，在Profile模式中可以把隔离层看作关于使用中间件的特定企业配置文件。

173

隔离层可以被看作层次化架构中的一个层[Mowbray 1997c]。与大部分其他层不同，这是非常薄的一层，不包含应用对象。一般而言，隔离层仅仅是作为把客户端和服务与一个或多个基础结构集成时的代理。Buschmann（1996）说明了Proxy（代理）模式。

### 6.4.10　对其他视角和规模的适用性

Vendor Lock-In对管理层的影响是在供应商产品发布的指挥下失去了对技术、对IT功能和对运行及维护预算的控制。Vendor Lock-In还会影响到风险管理。该反模式常常是在了解到供应商

产品要实现的新功能的情况下被接受的。不过，这些功能的交付时间往往比预期或需要的时间更晚，甚至根本就不会交付。

Vendor Lock-In通过要求开发人员具有深入的产品知识来影响他们。但是这些知识只是短时效的；它们会随着下一次产品发布而过时。因此，开发人员需要持续处于对产品功能、产品错误和产品未来的学习曲线中，而这是相当单调的工作。

### 小型反模式：Wolf Ticket（黄牛票）

**反模式问题**

信息系统标准的数量比起保证符合这些标准的机制的数量要多得多。只有6%的信息系统标准具有测试套件。大部分可测试的标准是用于编程语言——如FORTRAN、COBOL和Ada等编译器的。

如果产品声称具有开放性、符合实际上并没有实施方法的标准，它就是Wolf Ticket。该产品交付时采用专有接口，与公布的标准之间可能存在巨大的差别。关键问题在于技术的用户往往认为开放性会带来一些好处。实际上，标准对技术供应者在品牌认可上的作用比它能够为用户带来的任何益处都更为重要。

标准确实可以降低技术迁移成本，提高技术稳定性。但是，标准实现中的不同常会排除掉



它们本来可以带来的效益，如多供应商互用性和软件可移植性。而且，许多标准的规范过于灵活，无法保证互用性和可移植性；其他标准又过于复杂，产品中对这些标准有不完整和不一致的实现。很常见的是，不同供应商实现的是标准的不同子集。

Wolf Ticket对于事实标准（通过广泛使用和市场展示而建立的非正式标准）是很显著的问题。不过，某些事实标准没有有效的规范。例如，有一种可以通过商业购买获得的新生数据库技术具有多个专有接口来分别对应不同的供应商，它已经成为了一种事实标准。

**重构方案**

技术上的缺口导致了在规范、产品可用性、对标准的符合性、互用性、稳健性和功能方面的不足。封闭这些缺口才能交付那些包括实现有用的系统所必需的基础结构和服务的整个产品。在20世纪60年代，有一个被称为SHAPE的有经验的用户组建议在行业中对技术进行稳定，为计算机主机市场建立成功实现非Stovepipe System所需的整体产品。其结果是，大型机工作集成了硕果仅存的信息技术整体产品市场。

技术缺口成为了最终用户、集体开发人员和系统集成者要面对的非技术问题。非技术是对权力的使用，而技术缺口只有在用户先要求解决它们以后才会被有效地处理。例如，用户必须先要求保证具有适销性和“适用于目的”，商业供应者才会提供具有这些性质的产品。

草根非技术的核心策略是提升矛盾。通过在系统（例如技术市场）中传播对矛盾的了解，公司（这里就是技术供应者）才会解决这个问题。有效的非技术信息由3个要素构成。具有这3个要素时，该信息就很有可能会被媒体报导：

（1）信息必须是存在争议的。

（2）信息必须是反复出现的。

（3）信息必须是有趣的。

目前，我们正在对技术用户非技术进行初步研究。

175

现在需要的是能够支持关键业务系统开发活动的整体产品。允许建立关键业务系统的整体产品都具有5个关键服务：命名、交易、数据库访问、事务处理和系统管理。这些服务适用于所有领域的关键业务系统。命名服务是一项白页服务（从名字查找详细信息），允许获取已知对象的对象引用。交易服务是一项黄页服务（根据分类进行查找），通过根据属性获取候选服务来支持系统扩展。标准的数据库访问服务对获取和更新信息资源是不可缺少的。事务处理保证稳健地访问状态信息和在出现失败时有秩序地进行清理。系统管理对维护异类的软硬件是必需的。

目前，由于开发人员无法以稳健的、交互作用的形式来购买这些整体产品，他们不得不重新建立这些服务或构建Stovepiepe System。

**变化**

所有的计算机技术用户都可以参与改进他们当前正在使用的技术。要进行参与，只需要简单地给供应商打电话，提出疑问、抱怨和支持问题。要记住，对于简装产品，它的利润额度比答复电话和解决问题需要的资源更少。大多数供应商会跟踪支持问题，在他们产品的将来版本中做出相应的修改。他们通常是根据报告的问题的出现频度和紧迫性来确定修改的优先级。

**背景**

Wolf Ticket这个术语来自于俚语，指的是由投机者以非官方方式卖出的某项活动如摇滚音乐会的入场券。

176

反模式

## 6.5　Architecture By Implication（实现主导架构）

**反模式名称：** Architecture by Implication
**别名：** Wherefore art thou architecture？（架构在哪里？）
**最常见规模：** 系统层
**重构方案名称：** Goal Question Architecture（目标疑问架构）
**重构方案类型：** 文档
**根源：** 自负、懒惰
**不平衡的力量：** 复杂性管理、变化管理和风险管理
**轶事证据：** “我们以前这样建立过系统！”“没什么风险，我们知道自己在做什么！”

### 6.5.1　背景

德怀特·艾森豪威尔（第34任美国总统）曾说过，进行规划是重要的，但制定的计划是不合理的。另一个军人也说过没有任何计划在首次接敌后还有用。现代管理中的规划文化在某种程度上应感激兰德公司的创建者罗伯特·麦克纳马拉（1961~1968年担任美国国防部长）。在他的方法中，是带着投机的目的来生成计划，用于研究不同行动途径的潜在效益和效果。由于在系统开发中存在大量未知因素，IT系统规划必须更注重实效，并采取迭代的方式。

有一名专业的规划师说过，一名工程师的20%时间应该用于做规划。随着我们经验的增加，我们对这一论断的相信程度也在增长。通过规划来很好地组织工作后，生产率和效率都会得到极大的提高。不幸的后果是有很多机构试图把过多的规划活动形式化。规划在由个人来推动和利用
177 时最有效。时间管理专家传授的一个减少压力的关键要素就是通过规划让生活中的各种重要活动保持均衡。随着这种实践活动的成熟，时间管理系统的形式和使用方法越来越个人化。

美国国防部下属的系统集成公司的行政总监们成立了一个群组来回答“你的架构是什么？”这个问题。他们的目标是反映出系统开发活动的变化的本质。这些开发活动已经发展到复用已有的遗留构件和商业软件，不再进行绿地（greenfield）式[①]的定制代码开发（参阅本章的Reinvent the Wheel反模式）。

### 6.5.2　一般形式

本反模式的特点是开发中的系统缺乏架构规范（见图6-13）。通常，负责项目的架构师具有构建以前的系统的经验，因此认为文档是不必要的。这种过度的自信导致在影响到系统成功的关
178 键区域中风险剧增。下列某些区域往往会缺失架构定义：

① greenfield原指未开发的土地，在软件开发中指完全从头开发。——编者注

- 包括对语言和库的使用、编码标准、内存管理等在内的软件架构和规范。
- 包括客户端和服务器配置的硬件架构。
- 包括网络协议和设备的通信架构。
- 包括数据库和文件处理机制的持久性架构。
- 包括线程模型和信任系统集的应用安全架构。
- 系统管理架构。


图6-13　Architecture by Implementation常常形成在没有架构规划的条件下进行编码

### 6.5.3　症状和后果

- 缺乏架构规划和规范；对软件、硬件、通信、持久性、安全和系统管理架构的定义不足。
- 由规模、领域知识、技术和复杂性导致的隐藏风险随着项目的进展暴露出来。
- 由于性能不足、过度复杂、需求理解错误、可用性问题和其他系统特性导致项目将要失败或系统不成功。例如，大约1/3的系统在开发和运行中会遇到严重的性能问题。
- 不了解新技术。
- 缺乏后备技术和应急计划。

### 6.5.4　典型原因

- 没有风险管理。
- 管理人员、架构师和/或开发人员过于自信。
- 依赖于过去的经验，而这些经验与现实在某些关键区域有区别。
- 由于系统设计活动中的缺口导致隐含的和未解决的架构问题。

### 6.5.5　已知例外

在重复进行的方案中只在代码上有微小区别时（例如安装脚本），Architecture by Implication [179]

反模式是可以接受的。该反模式在新项目领域中作为确定已有技术是否可以转移到新领域的探索性活动也可能是有益的。

### 6.5.6　重构方案

Architecture by Implication反模式的重构方案要求以有组织的方式进行系统定义，并依赖于系统的多个视图。每个视图从一个系统利益相关者的角度对系统进行建模，这里的利益相关者可能是真实的也可能是假想的，可能是个体也可能是一群人的聚合。每个利益相关者负责一组高优先级的问题，而每个视图都代表了整个信息系统并回答了这些关键问题。这些视图包括一组图、表和规范说明，被连接到一起以保证一致性。一般而言，视图是轻量级的说明。架构文档的作用是交流架构性决策和其他问题的解决方法。文档应该易于理解，维护成本低廉。

有一个说法，只有完整理解一个架构的人才能成功地定义和实现它。不过，现实往往并不是这样，因为很多项目都采用了一些没有被很好理解的新技术。因此，从头开始建立良好的架构是一个迭代式的过程，大家都应该认识到这一点。起初的参考架构应该具备可以在第一个产品的开发期间被实现的强大策略。然后，可以用将来的参考架构版本以增量方式精炼它，并使用第一个产品或新产品的新版本来驱动这一过程。

使用视角来定义系统架构的步骤如下[Hilliard 1996]：

（1）定义架构目标。该架构必须达到什么目标？设计和实现必须让哪些利益相关者，包括真实的和假想的，感到满意？系统的前景是什么样子？我们目前处于什么位置，朝哪里前进？

（2）定义问题。必须解决哪些特定问题来满足利益相关者的要求？确定这些问题的优先级来帮助选择视图。

（3）选择视图。每个视图将代表系统架构的一幅蓝图。

180

（4）分析每个视图。从每个视角对架构定义进行细化。建立系统蓝图。

（5）集成蓝图。检验视图是否体现了一致的架构定义。

（6）追踪视图到需求。视图应该解决已知的问题，以发现架构规范中未解决的缺口。使用正式需求来验证架构。确定显著问题的优先级。

（7）对蓝图进行迭代。对视图进行精炼，直到解决了所有的问题和缺口。利用回顾过程来揭示所有剩下的问题。如果还有大量未解决的问题，应考虑增加新的视图。

（8）推广架构。做出明确的努力来和关键利益相关者，尤其是系统开发人员交流该架构。建立持久性的文档（例如视频教程）来在整个开发和维护生命周期中提供有价值的信息。

（9）验证实现。应该以“as-built（如同已建好）”设计的方式来表述蓝图。确定在蓝图和系统实现之间所有不同的细节。判断这些差异是否应导致修改系统或更新蓝图。更新文档以保证一致性。

我们把这种方法称为goal-question architecture（目标质疑架构，GQA），类似于软件测量中的目标质疑度量方法[Kitchenham 1996]。

### 6.5.7　变化

还有一些方法使用视角来考虑系统，其中一些使用了预定义的视角。大部分这类方法是开放式的，大家可以按照前述的方法来选择额外的视角。

RM-ODP（开放式分布处理参考模型）是一种广泛使用的、有效的分布式架构标准。它定义了5个标准视角，分别是企业视角、信息视角、计算视角、工程视角和技术视角[ISO 96]。它还从工程视角为分布式基础结构定义了一组有用的透明属性。

另一种方法是Zachman Framework（Zachman框架），它从数据、函数和网络的角度来分析系统架构[Spewak 1992]。每个角度都有多个层次的抽象，分别对应不同利益相关者群的规划需要。企业架构规划（Enterprise Architecture Planning）是基于Zachman框架的用于大规模系统的方法[Spewak 1992]。不过这两种方法都没有经过裁剪以适应面向对象系统的开发。 181

第三种方法，C4ISR-AF（指挥、通信、控制、计算机、情报、监视和侦察体系结构）被用于定义多种指挥和控制系统架构。C4ISR-AF的其中一个版本被用于其他类型的民用系统。该方法在帮助完全不同领域的架构师进行交流方面非常有益[Mowbray 1997b]。

第四种方法是4+1 Model View（4+1模型视图），它是一种基于视角的架构方法，受到如Rational Rose（已更名为IBM Rose Software Modeler）等软件工程工具的支持[Kruchten 1995]。这些视角包括逻辑视角、用例视角、过程视角、实现视角和开发视角。最后，GQA是对这些架构方法中所使用的内在方法的归纳[Hilliard 1996]。

### 6.5.8　示例

有一种常见但是不良的实践方法是在不定义视角的情况下进行面向对象建模。在大部分建模方法中，都存在视角模糊的情况。许多建模构造都包含实现细节，而默认的做法是混合实现和规范中的构造。

三个最基本的视角是：概念视角、规范视角和实现视角[Cook 1994]。概念视角从用户的角度来定义系统。通常它被称为分析模型。在这个模型中一般不会指出哪些被自动化了，而哪些没有自动化。画出这个模型是为了让用户可以向他人解释和保护自己的要求。规范视角只关心接口。ISO IDL是一种重要的标注方式，被严格限制于定义接口信息而不包括实现细节。接口和实现的隔离使得可以实现对象技术带来的多种重要效益，例如复用、系统扩展、变化、可替换性、多态和分布式对象计算。最后一个是实现视角，最好用源代码来体现它。使用面向对象设计模型可以对复杂的实现结构进行有益地补充，帮助当前的和未来的开发人员和维护人员更好地理解代码。 182

下面是Architecture by Implication反模式的另一个例子，其中的关键利益相关者对要构建的产品的经验没有形成整体的经验。项目的意图是交付一个基于微软DCOM（分布式组件对象模型）

的方案来提取遗留主机中的数据，根据业务规则对数据进行过滤，然后显示在Web页面上。然而，虽然项目经理是一个优秀的软件工程师，但是没有分布式对象技术（distributed object technology，DOT）的经验，而架构师则是曾帮助OMG生成Object Management Architecture（OMA，对象管理体系结构）的彻底的CORBA信奉者。让问题更为复杂的是，项目成员中了解DCOM的人非常少，只有不到10%。

此外，架构和后续的设计都是根据DOT世界中的OMA视图而不是DCOM来建立的。这导致试图在DCOM架构下交付CORBA服务。最终构成产品的一组构件没有DOT的一致性，性能也很差。而且，SI发现它由于缺乏标准化的方法而难以使用。最后，它在市场上失败了。

### 6.5.9　相关解决方案

Architecture by Implication反模式与Stovepipe System反模式的区别在于两者的范围不同。后者关注的是计算架构中的缺陷，尤其指出了对子系统API的不适当抽象是如何导致了脆弱的架构方案。Architecture by Implication反模式与之不同，涉及的是多个架构视角形成的规划缺口。

### 6.5.10　对其他视角和规模的适用性

该反模式显著增加了那些推迟重要决策直到出现失败的管理者的风险，那时再想亡羊补牢往往已经为时过晚。开发人员受到在系统实现中缺乏指引的困扰。他们必须在事实上承担起关键架构性决策的责任，但可能并不具备做出这些决策所必需的架构性视角。应该考虑接口设计决策在整个系统范围的后果，尤其是对系统可适性、一致的接口抽象、元数据可用性和复杂性管理的影响。

该反模式的另一个重要后果是延缓了对资源的分配。由于缺乏规划，重要的工具和技术构件在需要的时候可能并不可用。

183

#### 小型反模式：Warm Bodies（温暖身体）

**别名**

Deadwood（朽木）、Body Shop（美体小铺）、Seat Warmer（座位加热器）和Mythical Man-Month（人月神话）[Brooks 1979]。

**轶事证据**

“20个程序员中只有一个人……产出的软件是普通程序员的20倍。”

美国大约有2百万人从事软件管理和开发工作。目前还有20万个空缺的职位。这些数字显示出失业率是-10%。

**反模式问题**

有经验的程序员对软件项目的成功非常重要。所谓的英雄程序员的生产率尤其高，但是只有1/20那么少的人具有这样的天赋。他们产出的可工作软件比普通程序员要高出一个数量级。

大规模软件项目在许多行业中都很常见。这些项目雇佣了数以百计的程序员来构建一个企业系统；单个项目中有100~400个人并不是什么反常的现象。这些大型项目常常包括根据工作时数来计算的外包开发和合同付款。由于利润是人员工资的一个百分比，所以工作时数越多，利润就越高。系统需求在开发过程中总在改变和增加，所以即使一开始对项目要价过低也不会有多少风险；承包人会增加人手来解决不可避免的问题和新需求。Frederick Brooks的*Mythical Man-month*（1979）（《人月神话》，清华大学出版社2002年中译本）一书中说明了给进行中的软件项目增加人手所犯的错误。

**重构方案**

理想的项目规模是4个程序员；理想的项目持续时间是4个月[Edwards 1997]。软件项目和委员会会议一样受到群体动力学的影响（参阅接下来的Design by Committee反模式）。超过5个人的项目团队一般都会在群体协调方面遇到逐渐增加的困难。团队成员会难以制定有效的决策，难以维护共同的愿景。要鼓励团队集中注意力开始产出解决方案，设定一个近期的最终期限作为工作目标是很重要的。

我们认为超大型的项目是徒劳的。分别承担责任的小型项目团队更可能产出成功的软件。

**变化**

发现有天赋的程序员对软件密集的公司是重要的挑战。有些公司采取在招聘过程中进行测验的办法。这些测试类似于IQ测试。如果应聘者未能通过测试，他很可能会进入一个大型项目，和几百个其他程序员呆在一起。

与独立承包人和独立顾问合作是迅速获取编程能力的有效手段。在美国的某些地区，有数以百计的合同制程序员，可以通过电话或电子邮件联系到他们。与由于Warm Bodies小型反模式导致的项目失败和超期相比，这些合同制程序员可以用合理的速度产出大量的软件产品。

184

185

反模式

## 6.6　Design By Committee（委员会设计）

**反模式名称：** Design By Committee

**别名：** Gold Plating（镀金）、Standards Disease（标准疾病）、Make Everybody Happy（取悦大众）

**最常见规模：** 全球层

**重构方案名称：** Meeting Facilitation（会议推动）

**重构方案类型：** 过程

**根源：** 自负、贪婪

**不平衡的力量：** 功能管理、复杂性管理和资源管理

**轶事证据：** “骆驼就是委员会设计出的马。”“人多坏事。”

### 6.6.1　背景

面向对象常被看做发展了两代的技术。以数据为中心的对象分析是第一代面向对象技术的特征，而设计模式则是第二代的特征。第一代面向对象技术所持的思想是“对象就是你可以触及的东西”。这种看法的后果是实际上所有设计都完全是纵向的、互不相同的。在第一代面向对象技术中，人们相信一些未能被实践验证的假设。其中之一就是项目团队中应该是平等的，也就是说每个人都应具有相同的发言权，而决策是采用民主的方法来制定。这就导致了Design by Committee反模式。由于只有少数对象开发人员可以定义良好的抽象，少数服从多数的规则总是会导致抽象的弥散和过度的复杂性。

187

### 6.6.2　一般形式

复杂的软件设计是委员会过程的产物。它有过多的功能和变化，以至于要让任何一组开发人员在合理的时间范围内实现这些规范都是不可行的。由于过度复杂、含糊不清、约束过多以及其他的规范缺陷，即使可以进行设计，也不可能对整个设计进行测试。由于有太多的人参与建立设计和扩展，最终的设计会在概念上不够清晰。

### 6.6.3　症状和后果

- 设计文档过于复杂、不可读、不连贯或者有过多的缺陷。
- 设计文档篇幅过于庞大（数百甚至数千页）。
- 需求和设计缺乏收敛性和稳定性。
- 委员会议如同闲谈一样，很少讨论实质性问题，进程极其缓慢。大家串行地发言和工作。也就是说，只有单线程的讨论，大多数人在大部分时间中是没有产出的。

- 环境受非技术因素的影响而改变，在会议之外无法做出什么决策或采取行动，而会议过程排除了及时做出决策的可能性。
- 设计的功能没有优先级，对哪些功能是本质的要求，以及在首次交付中要实现哪些功能这两个问题也没有答案。
- 架构师和开发人员对设计的理解存在矛盾。
- 设计过程严重超支、超时。
- 必须雇用专家来对规范进行解释、开发和管理。也就是说，处理由委员会设计出的每项规范变成了专职工作。

### 6.6.4 典型原因

*“一群疯子在掌管着疯人院。”*

——Richard Rowland（美国Metro影业公司总经理）　188

- 没有指定项目架构师。
- 退化的或低效的软件过程。
- 不良的会议过程，标志是缺乏推动或低效的推动。会议变成了闲聊，声音最大的人获得胜利，讨论处于人员经验的最小公分母水平上。
- 镀金，也就是根据专有的利益在规范中增加功能。有很多原因会出现这种现象：适销性、已包含这些功能的专有技术的存在，或者在规范中为了将来可能的工作而投机性的置入功能。
- 试图让所有人都感到高兴，通过合并所有与会者的意见来让大家都满意。不过，不可能接受所有意见而不影响到复杂性管理。
- 会议期间有5人以上试图进行设计和编辑。
- 未确定明确的优先级和软件价值体系[Mowbray 1997c]。
- 各种关注点没有被隔离开，也未使用参考模型。

### 6.6.5 已知例外

Design by Committee反模式的例外很少见，只在委员会很小，只有大约6~10人时才有可能。人数更多的话就不太可能达成一致；少于6个人则对问题的理解和经验的范围就不够了。而且委员会往往应该是“飞虎队”，也就是在解决特定问题期间组织起来的一小组特定问题领域的专家。

### 6.6.6 重构方案

Design by Committee的解决方案的关键是改革会议过程。许多时候，大多数人习惯于忍受不良会议这个说法是正确的。因此，对会议过程即使是简单的改变也会对会议的产出率产生实质性的提高。会议产出率提高后，就有可能产生提高了质量、更为深思熟虑的方案。软件优化带来的效益一般小于一个数量级（2~10倍）。而会议产出率的提高则显著得多，往往可以达到数个数量

189 级（100倍）。我们甚至看到过产出率的提高超过10万倍。

首先，大部分会议室，尤其是宾馆的会议室的墙上没有钟。了解当前时间对会议进程是很关键的。应该指导与会者有效地管理对时间的分配。他们应该用“不超过25字”的摘要开始讲话，只在别人要求的时候才增加细节。贴出会议的目标和日程表并放置一座时钟让所有与会者都能看到，可以显著地改进会议情况。

其次，让会议成员回答“我们为什么在这里？”和“我们希望得到什么结果？”这两个问题对所有会议都是重要的。如果没有准备会议计划，让大家首先从这两个问题开始再继续生成需要的结果，这种做法尤为重要。

另一个重要的改革是明确分配软件过程中的各种角色：所有者、推动者、架构师、开发人员、测试人员和领域专家。所有者是负责该软件开发的管理者。他要制定有关整个软件过程的战略决策，并邀请和组织其他与会者。会议开始时，过程所有者要设定目标，并建立如何对会议结果进行处理的预期方法。例如，可以把会议期间做出的决策简单地看做是一些建议，也可以完全按照讨论的方法来进行实现。

推动者负责会议的运转。他的责任是对会议过程进行控制，而其他参与者则负责技术内容。如果需要做出有关会议过程的关键决策，推动者应寻求过程所有者的帮助。

“我的特长是在别人出错的时候做正确的事。”

——萧伯纳（英国剧作家）

架构师是软件项目的高级技术领导。他控制对架构文档的编辑，可能还负责关键的系统层边界，例如子系统应用编程接口。每个开发人员通常负责单个子系统和单元测试。测试人员负责对规范的质量进行监控，并进行增值测试，例如集成测试、可移植性测试和压力测试。领域专家向过程输入关键的需求，但可能不会参与到开发的所有方面。

有3类会议过程：发散的、收敛的和信息共享的。在发散的过程中，会生成以后要利用的意见。在收敛的过程中，会做出代表大多数人意见的选择或决策。信息共享则可以包括报告、教学、
190 编写文档和回顾。

参与每种会议过程的人数由推动者进行管理。需要编写文档、突出重点或绘图的创新性过程应限制于不超过5个人的突破团队。超过5个人的群组虽然在创新性过程之后对结果的评审和集成中很成功，但是在创新性任务中的效率较低。高产出率的会议包含许多并行的过程和突破组与评审组之间的频繁迭代。对推动者的一项关键挑战是鼓励大家建立在单线程的讨论和并行工作之间进行切换的范例。

大部分会议的主要目的是解决问题。通用的问题解决方法开始于一个收敛的过程：定义群组要解决的问题并确定它的范围。发散的过程用于确定可选的解决方案。可能还需要通过信息共享

来探索选定的可选方案的细节和效果。最后，使用收敛的过程来在选项中做出选择。

有一种被称为Spitwads[Herrington 1991]的会议过程相当有效。这是一种通用的会议程序，我们在很多场合使用过它并获得了很好的结果。

（1）提出问题。推动者提出一个问题让大家快速思考。问题被写在一个屏幕或白板上，以免发生误解。在开始思考之前要询问大家是否要对问题做出什么修改。比较典型的问题有："我们有哪些办法来提高系统的性能？"以及"尚未解决的需求中最重要的是什么？"

（2）安静地写下答复。与会者在一样的纸片上写下对问题的答复。每个答复都写在一张单独的纸上，并且只使用简短的语句。

（3）扔纸团。在与会者写下意见之后，推动者就指导他们把那张纸揉成一团，扔到房间另一边的一个容器中——纸板箱或篮子的效果最好。像投篮那样，推动者应该鼓励与会者在这项活动中找到一些乐趣。

（4）随机读出答复。把那些纸团任意地分配给与会者，让他们一个一个大声读出纸条上的意见，记录到白板中。可以同时用两个白板记录者来加速这个过程。最后把白板贴在墙上让所有与会者都能看见。

（5）达成共识。对白板上的意见进行编号。然后推动者会问大家是否不能理解某些意见。如果有的话，就鼓励大家提出定义。如果某个意见无法被定义，最安全的做法就是把它去掉。 191

（6）去除重复。推动者让大家确定出重复的意见或应该合并的意见。与会者确定那些要改变的意见的编号。如果存在异议，就应该否决这个改变（有一个通用的简化文档编辑工作的方法：如果有人反对做出修改，就不要接受这个修改）。

（7）确定优先级。指导大家安静地选择出列表中最佳意见的编号。可以选出不止一个意见。推动者要求大家对列表中的意见逐个进行投票（只举手，不讨论）。

（8）讨论。过程已经完成了。大家讨论具有最高优先级的选项，并提出接下来的行动。

### 6.6.7　变化

Railroad反模式（也称为Rubber Stamp，橡皮图章）是Design by Committee的一种变形。在这种反模式中，一个非技术联合控制了开发过程，强迫采用存在严重缺陷的设计。Railroad反模式往往是受委员会中许多成员的共同商业利益所驱动。通过采用不完整的、有缺陷的规范，技术细节可以被有效地隐藏在软件中。这样，联合成员开发的软件就成为了事实标准，而不同于编写好的规范。联合之外的某些开发人员甚至会试图去实现那些"错误功能"，导致时间和金钱的浪费。

### 6.6.8　示例

Design by Committee反模式的两个典型示例来自软件标准化领域：SQL和CORBA。

#### SQL

SQL在1989年成为国际标准。最初的SQL89是一份只有115页的小文档，体现了对该技术高

192 效的、最小化的设计[Melton 1993]。实际上所有关系数据库产品都完整地实现了该规范。1992年，SQL的第二个版本得以标准化，它进行了大量的扩展，形成了580页的文档。每个产品都用独特的“方言”来实现SQL92规范，没有几个产品实现了整个规范。下一版的SQL被称为SQL3，很可能会有数千页长。负责设计的标准委员会像自助餐那样增加了大量新功能，把概念扩展到远远超过最初的意图。新功能中包括面向对象扩展、空间地理扩展和时间逻辑扩展。任何产品都不太可能完整实现SQL3，也不太可能有两个产品会以可移植的方式实现相同的子集。在这个典型的Design by Committee反模式中，SQL标准变成了先进的数据库功能的垃圾堆积场。

ODBC（Open Database Connectivity，开放式数据库连接）和JDBC（Java Database Connectivity，Java数据库连接）技术提供了引人注意的解决SQL收敛性的方案。两者分别定义了基于动态查询的标准API，这些动态查询是在运行时提交和解析的查询语句。由于ODBC和JDBC为客户端定义了查询接口和查询语言，它们就提供了对产品特定的数据库功能的隔离。客户端可以透明地访问多个数据库产品。ODBC通过SAG（SQL Access Group，SQL访问组）这个软件行业协会成为了行业标准。微软独立于SAG之外开发了ODBC规范，然后提交给SAG。SAG迅速接受了这个规范，使它成为了事实上的行业标准。推动使用专有解决方案的供应商无法取代这种非常高效的技术，它已经被数据库供应商和数据库工具开发人员普遍地支持。

### CORBA

CORBA标准是在1991年被软件行业所采纳的。最初的文档只有不到200页，即使是不熟悉OMG过程的人也很容易阅读它。1995年，发布了修订的CORBA2规范，对它进行了大量的升级和扩展，包括C++和Smalltalk映射、Interface Repository（接口仓库）以及IIOP（Internet Inter-ORB Protocol）。尤其是CORBA2中的C++映射包括繁杂的细节，ORB（Object Request Broker，对象请
193 求代理）供应商根本不可能完全一致地实现它们。其中的一些供应商会反复修改由IDL/C++编译器生成的API。CORBA2中对BOA（Basic Object Adaptor，基本对象适配器）进行了重大修改。POA（Portable Object Adaptor，可移植对象适配器）于1997年被采用以替代CORBA中的这个部分。BOA已经存在于使用中的产品中，由于POA与之相重复，那些已经取得成功的供应商就缺乏升级他们的产品的动机。而且，由于对基本的底层结构投入了如此多的注意力，用户的某些更为迫切的要求只获得了较低的优先级。

OMG有一个用于确定和定义要采用的技术的替代过程，它使用这个过程提出了CORBA-facilities Architecture。其他特别工作组和最终用户企业复用了这个过程来定义他们的架构和路线图。本书的作者们称该过程为“Lost Architecture Process（丢失的架构过程）”，因为迄今为止该过程尚未被文档化。

Lost Architecture Process是简化过的架构定义程序，包括下面的步骤。可以用机构特定的过程如内部审查和远距离研讨会来替换其中的OMG过程，让它适应特定企业的需要。

（1）发布信息需求（RFI）。RFI的目的是调查OMG内部和外部相关群体的意见。所有机构的相关群体都被要求提交他们的需求、架构输入以及对相关技术的说明来帮助规划过程。

（2）评审RFI答复。特别工作组评审收到的所有答复（通常不会超过一打）。这项工作将会完成过程的调查和数据收集阶段。一旦特别工作组完成了对输入的评审，他们的职责就转变到了定义架构。RFI过程在心理学上是一个重要的步骤，因为它把一组未定义的利益相关者和隐含的需求转换到了会议参与者的职责中，他们将据此定义架构，画出路线图。

（3）确定候选服务和设施。在白板中列出候选服务，贴在会议室中供与会者评审。在RFI评审过程中就可以开始建立列表，还可以通过快速思考环节给这些列表扩展额外的设施。重要的是获得所有的意见，然后对列表进行精选，挑出那些重复的和重叠的服务。

（4）启动初始RFP过程。可能在确定出的服务中至少有一项会是RFP（Request For Proposal，征求意见稿）发布的明显候选者。很可能某些供应商团队参加会议的目标就是在规范中采用特定的技术。可以分离出这个团队来处理初始RFP并启动特别工作组的采用过程。该过程的权衡阶段将定义其他的服务和设施。

194

（5）画出架构。一小支分离的群组将对列出的服务进行分类，以方框图的形式显示出服务的层次和横向轮廓。显然，该群组中至少要有一个人熟悉参考模型绘图的技巧。这个架构参考图是对列出的服务的有用抽象，将是架构文档中的重要图示。图中要列出所有的服务和设施。

（6）定义基本服务。群组把这些服务分成小包，以视图中的符号列表的形式进行定义。大家按照小组工作或者独立工作，以这种形式定义这些服务。由整个特别工作组对结果进行评审和讨论，这时可以建议和讨论对符号列表中的用语做出补充和修改。

（7）编写服务定义。使用RFP模板来编写每一个定义好的服务。利用会议之间的时间来处理编写定义的工作。这时，必须指定一名架构文档编辑（最好只有一人）来负责接受所有输入，建立文档草稿。

（8）草拟文档。文档编辑集中所有提交的服务定义和架构图来建立文档草稿。编辑可能会增加对总体架构的样本说明以便文档可以为自身提供支持。文档中还包含服务的表格和分类来汇总和抽象服务定义中的内容。

（9）过程评审。每次会议都要对更新的架构文档草稿进行评审。会上将提出和讨论在下一次修订中要做出的编辑改动。如果有缺失的部分或服务详细说明，就分配给群组成员独立地处理，然后由编辑来合并。

（10）定义路线图。路线图是关注于优先级和进度表的架构配置文件。有一些关键准则可以建立优先级和定义进度表。这些准则包括行业中对该技术的需要程度、技术之间的依赖性、技术适用的广泛程度以及特别工作组的工作量。定义路线图是整个过程中的关键部分，让成员群组可以组织资源，规划与该技术的采用过程相关的研究。

195

（11）引导批准过程。经过数次评审过程的迭代后，可以提出动议把架构和路线图发布为1.0版。在1.0版后还可以进行进一步的修改，但是通过这一动议意味着特别工作组在初始架构上达成了一致。

### 6.6.9　相关解决方案、模式和反模式

Kyle Brown在Portland Patterns Repository网站上贴出了一个Design by Committee反模式的版

本。该模式使用了不同的模板，完全集中在说明有问题的方案而不是重构方案上。本书与之不同，在每个反模式中都包括了一个重构方案，因为我们注意到，苏格拉底被处以死刑的原因在于他只揭露社会矛盾却没有提出任何建设性意见。

### 6.6.10　对其他视角和规模的适用性

Design by Committee反模式对开发人员的影响在于大家会期望开发人员实现高度复杂而模糊的设计，这种环境会有很大的压力。开发人员可能会发现需要暗中使用更现实的设计方法来破坏委员会的要求。

管理人员受到这个反模式的影响是由于过度的复杂性会导致项目风险的显著增加。与之相应，项目进度和预算很可能会由于实验室中发现这些设计所带来的后果而显著增加。

在系统层，如果不需要任何变化（多重配置），而且建议的实现进度被扩展30%以上，交付一个根据Design by Committee规范建立的系统也许是合理的。大多数开发人员只能处理平台、数据库和功能集等方面的少量变化。

196


图6-14　Swiss Army Knife：设计者包括了除了厨房水槽以外他们能想到的所有东西！

## 小型反模式：Swiss Army Knife（瑞士军刀）

### 反模式问题

Swiss Army Knife反模式也被称为Kitchen Sink（厨房水槽），是指过分复杂的类接口（见图6-14）。设计者试图提供该类所有可能的用途。为此，他在为了满足所有可能的需要而进行的无用尝试中增加了大量的接口签名。Swiss Army Knife反模式的现实例子在单个类中会提供数十个到数千个方法签名。接口中缺乏重点表明设计者对该类可能缺乏清晰的抽象，或不了解它的用途。Swiss Army Knife在商业软件接口中非常普遍，因为供应商们试图让他们的产品可以适用于所有可能的应用。

197

该反模式存在问题的原因在于它忽略了复杂性管理这一作用力。也就是说，其他程序员难以理解这个复杂的接口，而且这样的接口让该类的使用意图变得模糊不清，即使在简单的情况中也是如此。复杂性导致的其他后果还包括在调试、文档编写和维护中的困难。

### 重构方案

在软件开发项目中常常会遇到必须使用复杂的接口和标准。因此，重要的是定义使用这些技术的协定以免破坏对应用的复杂架构的管理。这被称为建立一个配置文件。配置文件是文档化的协定，解释了应如何使用一项复杂的技术。配置文件往往是用于技术细节的实现计划。通过使用配置文件，两个独立的开发人员可以使用相同的技术，让获得可互用的软件成为可能。

软件接口的配置文件定义了要使用的方法签名的子集，它应该包括参数取值的协定。也就是说，配置文件指明了在每个参数中可以传递的值。此外，在定义使用该接口的应用的动态行为时，可能也会需要配置文件，包括执行顺序、方法调用和异常处理方面的说明和规范。

### 变化

Swiss Army Knife与The Blob反模式的不同在于在一个设计中可能存在多个Swiss Army Knife。此外，Swiss Army Knife的内涵是在设计人员为了满足该类可以预见的所有需要而进行的徒劳尝试中暴露出了复杂性。而The Blob是单个对象独占了系统中的处理或数据。

198

反模式

## 6.7　Reinvent The Wheel（重新发明轮子）

**反模式名称：** Reinvent the Wheel
**别名：** Design in a Vacuum（真空设计）、Greenfield System（绿地系统）
**最常见规模：** 系统层
**重构方案名称：** Architecture Mining（架构挖掘）
**重构方案类型：** 过程
**根源：** 自负、无知
**不平衡的力量：** 变化管理、技术转移管理
**轶事证据：**“我们的问题是独一无二的。”软件开发人员对其他人的代码一般只有很少的了解。每个程序中，即使是对以源代码形式被广泛使用的软件包也很少会有一个以上有经验的开发人员。

实际上几乎所有系统开发活动的完成都是隔离于与之具有重叠功能的项目和系统的。复用在大部分软件开发机构中都很少见。最近，在对超过32个面向对象软件项目的研究中，研究人员发现几乎根本没有成功复用的证据[Goldberg 1995]。

### 6.7.1　背景

软件复用和设计复用是具有显著差异的范例。软件复用要求建立可复用构件库、获取这些构件和在软件系统中集成这些构件。典型的结果是围绕着系统外围的适度复用以及为了集成构件而进行
199 一些额外的软件开发。设计复用则是在多个应用系统中复用架构和软件接口。它要求确定在多个应用系统中都具有作用的横向构件。设计复用还支持在不进行额外集成开发的条件下对横向构件进行软件复用。因此它是更为有效的方法，因为可以利用可复用构件得到软件系统中更多的部分。

绿地这个词（在Reinvent the Wheel反模式的别名Greenfield System中）起源于建筑业。它指的是一块新建筑场地，在这个场地中没有遗留建筑来限制新建筑的架构。

### 6.7.2　一般形式

虽然已经有数个系统具有重叠的功能，定制的软件系统仍然是从头建立的。该软件过程对单个系统采取了“绿地”（从头建立）开发。由于自顶向下的分析和设计会产生新的架构和定制的软件，软件复用受到了限制，事后才会想办法实现互用性。

大部分当今的软件开发方法都假定开发人员是从头开始建立定制的软件，而且他们是在孤立地构建单个系统。这被称为绿地系统假设。绿地系统不可避免地会成为缺乏互用、扩展和复用潜力的烟囱。绿地假设对现实中的大部分软件开发问题都是不合适的。在这些问题中存在着遗留系

统，与它们的互用是对许多新系统的重要需求。绿地假设还忽略了以因特网自由软件和商业软件形式存在的重要的可复用软件资产。

### 6.7.3　症状和后果

- 每次设计一个系统，没有准备复用和互用而形成的封闭系统架构——架构和软件。
- 对商业软件功能的复制。
- 不成熟、不稳定的架构和需求。
- 对变化管理和互用性的支持不够充分。
- 在架构足够成熟可以支持长期系统开发维护之前，扩展的开发周期产生失败的原型和死胡同（dead-end）原型。
- 对风险和成本的管理很差，导致超时和超支。
- 无法向最终用户交付需要的功能；花大量的精力来复制已有系统中已经提供的功能。 200

### 6.7.4　典型原因

- 软件开发项目之间没有交流和技术转移。
- 缺乏包括架构挖掘和领域工程管理的明确的架构过程。
- 采取了绿地开发，也就是说开发过程假定会从头开始建立系统。
- 缺乏对计算视角的企业级管理，导致在每个系统中分别采用不同的接口。

### 6.7.5　已知例外

Reinvent the Wheel反模式对研究性环境是合适的。而在开发人员具有不同技能水平，工作于遥远的不同场所时，该反模式也适合于一般性开发来降低这种环境下的协调成本。

### 6.7.6　重构方案

成功的面向对象架构是稳健的、不依赖于产品的、可复用、可扩展的，而架构挖掘是一种可以迅速建立这种架构的方法。大部分面向对象设计都假定会随着开发过程的进展产生设计信息。在自顶向下的过程中，设计信息来自于需求，而需求可以被表现为用例和面向对象分析模型。需求驱动的架构设计被称为架构培养。在螺旋式开发过程中，每次迭代都会产生设计信息。随着螺旋式过程的进展，架构师会随着对应用问题的更多了解而获得新的设计信息。完全可以说这些方法重新发明了它们的大部分设计信息。

对大部分信息系统应用和问题而言，都有前驱设计存在。这些设计以遗留系统、商业软件、标准、原型和设计模式的形式存在。经验证明，对任何给定的应用系统都不难发现半打以上的前驱设计。有价值的信息被埋藏在先前存在的设计中，这些信息让以前的架构师可以构建有用的系统。提取这些信息用在面向对象架构中的过程被称为架构挖掘。 201

对某些复杂的设计问题，挖掘可能适用于应用层。有些时候，利用已有的专家经验可能比在

不了解已有解决方案的情况下建立新代码更廉价，风险也更低。架构挖掘可以适用于企业层，但在全球层则没有那么有效，因为可以访问的信息资源减少了。

挖掘是自底向上的设计方法，利用的是来自于可用的实现中的设计知识。挖掘也可以利用来自于自顶向下的设计过程的设计输入，所以挖掘得到的设计中可以既有自顶向下的可追溯性，也有自底向上的真实性。

“不成熟的艺术家进行模仿。成熟的艺术家直接偷取（思想）。”

——Lionel Trilling（美国文学评论家）

在开始架构挖掘前，必须确定一组与设计问题有关的代表性技术。有多种方法可以完成技术确定工作，例如搜索文献、与专家会谈、参加技术会议、上网查找，等等。应该在所有可用的资源中进行探索。

架构挖掘的第一步是对每种代表性技术进行建模，以便产生相关软件接口的规范。我们建议使用OMG IDL作为接口标注方法，因为它很简练而且与实现细节无关。OMG IDL也是一种良好的目标架构设计标注方法，因为它与开发语言和平台无关，对分布也是透明的。用同一个标注方法对所有内容建模可以为设计的比较和折中建立良好的基础。

建模时，重要的是把系统当作已经构建好的形式进行说明，而不是说明意图的或期望的设计。相关的设计信息往往并没有被记录为软件接口。例如，正在找寻的某些功能可能只有通过用户界面才能进行访问。其他的关键设计教训可能也没有被文档化，捕捉这些信息也是有益的。

在挖掘的第二步中，对设计进行归纳来建立通用接口规范。这一步更多地要求艺术能力而不是科学能力，因为目标是为目标架构接口建立一个起始的“稻草人”规范。建立代表性技术的最
202 小公分母设计通常是不够的。归纳出的接口应该包括捕获到共同功能后“优化杂交”出的方案，以及由特定系统产生的某些独特方面。如果这些独特方面在目标架构中建立了有价值的功能或者代表了已知的系统发展方向，就应该包括它们。对代表性技术的稳健分类会包含对目标系统的可能发展方向的指示器。

这时，就可以考虑把自顶向下的设计信息作为一部分输入。自顶向下的信息通常比自底向上的信息具有更高的抽象水平。要让这些差异取得调和，需要做出一些重要的架构折中。

挖掘过程中的最后一步是对设计进行精炼。可以用架构师的判断、非正规的走查、评审过程、新需求或额外的挖掘研究来驱动精炼过程。

### 6.7.7　变化

在软件开发机构内部难以形成软件复用。在对数十个面向对象应用开发项目的调查中，Goldberg和Rubin发现没有明显的复用[Goldberg 1995]。即使有成功的复用，内部复用在成本方面

带来的效益通常也低于15%[Griss 1997]。行业经验表明内部复用扮演的主要角色是作为在转售软件中的投资。这时，巨大的数量让潜在节约的资金变得显著，而且复用可以缩短投入市场的时间，并支持产品定制。

另一方面，我们认为复用是很普遍的，不过是以不同的形式：对商业软件的复用和对自由软件的复用。由于具有更大的用户基础，商业软件和自由软件常常比定制开发的软件具有高得多的质量。对于有许多应用软件要依赖的基础结构构件，这种质量上的提高可能会关系到项目的成败。商业软件和自由软件在不经修改就使用时可以减少维护成本，而且在出现升级版本时可以很容易替换。

### 6.7.8　示例

架构挖掘要求研究已有的软件架构，对现存系统的软件接口尤其感兴趣。在分布式对象设计模式培训课上，我们使用下面的例子来说明架构挖掘中需要的方法和折中。下面是两个进行空间地理信息分类查询的已有系统的软件接口：

203

```
// ISO ODP IDL
 module Legacy1 {
   struct GeoCoords {
     double lat, lon; // degrees
 };
 typedef string QueryId;
 typedef sequence<string> NameList;
 typedef sequence<string> QueryHitList;
 struct QueryResults {
    NameList attribute_names;
    QueryHitList query_hits;
 };
 interface CA {
   // initiate query, get query results id, don't wait for query
   // completion
   QueryId boolean_query(
     in string boolean_query_expression); // Boolean Query
Syntax (BQS)
  // initiate query, get query results id, don't wait for query
  // completion
  QueryId point_query(
    in string boolean_query_expression,
    in GeoCoords point_geo_location);
// find out if query is finished
boolean query_finished(in QueryId query_result_identifier);
// get query results (after query is finished)
void get_results(
    in QueryId       query_result_identifier,
    in unsigned long number_of_hits_to_return,
```

```
      out unsigned long number_of_hits_remaining,
      out QueryResults product_records);
  };
};
```

第二个系统的软件接口如下：

```
module Legacy2 {
  struct GeoCoords {
    double lat, lon; // degrees
  };
  typedef sequence<GeoCoords> GeoCoordsList;
  typedef sequence<string> NameList;
  typedef sequence<string> QueryHitList;
  struct QueryResults {
    NameList     attribute_names;
    QueryHitList query_hits;
  };
  interface CA {
    // make query, get query results
    QueryResults boolean_query(
    in string boolean_query_expression); // Boolean Query
Syntax (BQS)
    // make query, get query results
    QueryResults polygonal_query(
    in string boolean_query_expression,
    in GeoCoordsList polygon_vertices);
  };
};
```

204

架构师在负责维护相应系统的开发人员的辅助下研究前面代码示例中的接口。他特别注意这些文档化的接口是如何被使用的，以及在文档和系统的实际使用中的不同。架构师要理解所有操作及参数的细节。下面是产生的“优化杂交”接口：

```
module BestOfBreed {
  struct GeoCoords {
    double lat, lon; // degrees
  };
  typedef sequence<GeoCoords> GeoCoordsList;
  typedef string QueryId;
  typedef sequence<string> NameList;
  typedef sequence<string> QueryHitList;
  struct QueryResults {
    NameList     attribute_names;
    QueryHitList query_hits;
  };
  interface CA {
    // initiate query, get query results id, don't wait for query
    // completion
```

```
    QueryId boolean_query(
        in string boolean_query_expression); // Boolean Query
Syntax (BQS)
  // initiate query, get query results id, don't wait for query
  // completion
  QueryId point_query(
    in string boolean_query_expression,
    in GeoCoords point_geo_location);
  // initiate query, get query results id, don't wait for query
  // completion
  QueryId polygonal_query(
    in string boolean_query_expression,
    in GeoCoordsList polygon_vertices);
  // find out if query is finished
  boolean query_finished(in QueryId query_result_identifier);
  // get query results (after query is finished)
  void get_results(
    in QueryId        query_result_identifier,
    in unsigned long number_of_hits_to_return,
    out unsigned long number_of_hits_remaining,
    out QueryResults product_records);
  };
};
```

205

请注意优化杂交后并没有把原始接口中的所有细微差别都结合进来。设计并不是原始接口的简单的最小公分母，也不是对两组接口及其所有功能的直接合并。架构师要归纳挖掘的系统中的共同功能，然后选择性地结合每个系统中的独特功能——某些可以预见是共同接口将来的需要。其他的独特功能仍然是系统特定的，不属于通用架构定义。

### 6.7.9　相关解决方案

在Mowbray 1995中对架构挖掘和归纳共同接口在复杂性管理方面的影响进行了分析。架构挖掘是重复出现的解决方案，可以解决很多由Stovepipe System引发的问题。它也是定义领域特定的构件架构的方法之一。

### 6.7.10　对其他视角和规模的适用性

Reinvent the Wheel反模式以增加投入市场时间和让功能低于最终用户预期的形式让管理人员处于更高的风险中。复用可以带来的可能效益是节约15%~75%的开发成本，缩短投入市场时间2~5倍，以及将缺陷减少5~10倍[Jacobson 1997]。

206

**小型反模式：The Grand Old Duke of York（约克老公爵）**

**别名**

Everyone Charges Up the Hill、Lack of Architecture Instinct（缺乏架构本能）和Abstractionists

versus Implementationists（抽象派对实现派）。

**轶事证据**

专家指出5名软件开发人员中只有1人能够定义良好的抽象[Mowbray 1995]。一名软件架构师在听到这种说法后的回答是："更像是50人中才有1人。"

**背景**

抽象派这个词来自艺术界，抽象派艺术家就是通过非写实的艺术作品来表达感情和态度的表现主义艺术家。抽象派在我们这个行业中是指具有架构本能的架构师或软件开发人员。

**反模式问题**

编程技能并不等同于定义抽象的技能。软件开发中似乎有两群不同的人：抽象派和与他们相对的人（我们称他们为实现派）[Riel 1996]。抽象派对讨论软件设计概念而不涉及实现细节感到很满意。如上所述，他们具有架构本能，也就是定义和解释良好的软件抽象的能力。另一方面，实现派常常需要看到源代码示例才能领会抽象概念；他们不是很善于定义让其他开发人员容易理解的新抽象。

很多面向对象过程是平等主义的；通过会议过程做出的设计决策是以多数人一致的原则被认可的（参阅本章前面的Design by Committee反模式）。根据专家的研究，实现派与抽象派的人数比大约是4:1[Mowbray 1995]。这样，抽象派的票数非常不幸地低于实现派。由于只有很少的开发人员认识到良好的抽象的重要性，它们往往会被破坏。首要后果是软件过于复杂，使得系统难于开发、修改、扩展、文档化和测试。由于未能使用有效的抽象原则，软件可用性和系统维护都受到了影响。

平等主义的面向对象开发方法在实践中同样低效；它就像让一群人同时往一座山上冲。每个人都要理解和遇到众多的决策，即使有一些有经验的成员，群体的总体经验也被最小公分母所降低，并被交流过程限制住。

**重构方案**

更有效的方法涉及软件开发机构中数个不同的角色。架构师是对大部分关键技术都具有大量经验的抽象派。他们推动最终用户和开发人员之间的交流；他们负责复杂性管理，确保系统的可适性。为了达到这个目的，架构师要管理计算架构或系统层编程接口。

构件开发人员是具有丰富经验的程序员。他们使用诸如C、C++和Java之类的系统编程语言建立软件基础结构和可复用的软件构件。

应用开发人员是集成这些构件，建立可用系统的其他程序员。他们主要使用诸如Visual Basic、JavaScript、Python、Tcl和Perl的脚本语言。脚本语言处于更高的层次，这一本质让更多的编程技能都能够提供产出。

**变化**

软件开发角色的特殊化表明在构建安全、有效的软件系统时存在多种学科。认证是建立和确认职业能力的潜在机制。认证对于美容师、司机、律师和注册公共会计师等多种职业都是必需的。为什么不对软件架构师进行认证呢？

207

208

# 第7章 软件项目管理性反模式

在现代工程型职业中，超过一半的工作涉及人员之间的交流和解决人员问题。管理性反模式指出了这些方面的问题会对软件开发过程产生破坏性影响的一些关键场景。

## 7.1 管理角色的转变

技术管理者的角色正在发生着变化。在无处不在的电子邮件和内联网面前，管理者主要是机构中的信息传递者。过去，是通过管理链来跨越组织结构边界传达信息，而在电子化的机构中，可以跨越空间、时间和组织边界进行无缝的交流。

传统上，管理层扮演的关键角色之一是批准规则和过程的例外。但是对组织结构的业务流程再造（business-process reengineering，BPR）显著改变了这一角色。进行再设计之前，组织边界强加了常常会降低生产率的遗留业务规则。而在再设计后的机构中，消除了影响生产率的边界，大家能够在无需管理层干预的条件下解决问题。




呆伯特（United Feature Syndicate公司授权）

但是，管理者在软件开发中仍然在以下领域扮演着一些重要的角色：

- 软件开发过程管理。
- 资源管理（人员和IT基础结构）。

- 外部关系管理（例如客户和开发伙伴）。

## 死亡行军项目

我们并不是第一个论述软件项目中的矛盾和缺陷的人[Webster 1995，1997]。Yourdon用死亡行军项目（death march project）来描述做出了不切实际的承诺的项目[Yourdon 1997]。他把目标或资源范围超出合理标准50%以上的项目定义为死亡行军项目。这意味着该项目满足至少一个下列的条件：

- 进度表比需要的时间短了50%。
- 人员规模只有必需规模的一半。
- 预算低于需要50%。
- 功能数目比资源相当的成功项目多50%。有些愤世嫉俗者甚至认为所有软件项目都是死亡行军项目。

我们赞成Yourdon的大部分观点，只有一个明显的例外。Yourdon引用了Scott Adams（呆伯特卡通画的作者）的一句话，声称死亡行军项目的根源是“大家都是笨蛋。”我们的观点是由于大家不是圣人，所以自然都会有过失。只有认识到自己在产品开发中犯下的过失才能克服它们。参
210 与产品开发的人越多，辨识错误（过程缺陷、角色缺陷、软件缺陷等）和纠正错误的任务中的复杂性就越高。

管理性反模式的目标是建立新的警觉，让你可以提高取得成功的可能性。管理性反模式描述了软件项目是如何被人员问题、过程、资源和外部关系所削弱的。这些反模式还说明了一些对这些问题的最有效解决方案。

我们很同情那些承担充满压力的软件项目的开发人员。我们更同情那些要承担项目失败的可怕后果的软件管理者[Yourdon 1997]。好管理者会帮助缓和整个软件开发机构中的压力水平。其后果是他们中的许多人就要独自承担大量的压力。除了本书中讨论的反模式解决方案之外，我们还推荐采用时间管理培训来让软件专业人员有效地学习如何管理压力。

## 7.2　管理性反模式摘要

下面的摘要是对管理性反模式的概述，还包括对与主要反模式一起讨论的管理性小型反模式的说明。

**Blowhard Jamboree（吹牛者狂欢）**：所谓的行业专家的意见常常会影响到技术决策。在大众媒体和非公开出版物上经常会出现批评特定技术的争议性报告。除了承担技术职责，开发人员还要花过多的时间来解答管理者和决策制定者由于这些报告而产生的顾虑。

**Analysis Paralysis（分析瘫痪）**：在分析阶段竭力争取完美和完善常会导致项目完全停滞和对需求/模型的过度“拷问”。重构方案中包括对迭代增量式开发过程的说明，这种过程可

以推迟进行详细分析直到需要这些知识的时候。

Viewgraph Engineering（**视图设计**）：某些项目中，开发人员会一直在准备视图和文档而不是开发软件。管理层从未获取合适的开发工具，而工程人员没有替代方法，只能使用办公自动化软件来生成伪技术图表和报告。

211

Death by Planning（**规划致死**）：对软件项目的过度规划导致造成后续问题的复杂进度表。我们解释了如何规划一个合理的软件开发过程，包括在规划中结合已知的事实，以及增量式的重规划。

Fear of Success（**成功恐惧症**）：当成功已经触手可及时，常会出现一个有趣的现象。有些人开始强迫性地担心各种可能会出错的事情。对专业能力的不安全感在这时浮出水面。

Corncob（**玉米棒子**）：某些难以相处的人往往会阻碍或偏转软件开发过程。应对这些Corncob的方法是通过不同的战术层、战役层和战略层组织行为来处理他们的日程表。

Intellectual Violence（**智力暴行**）：当某些理解了某种理论、技术和术语的人在会议中使用这些知识来胁迫他人时，就是出现了Intellectual Violence。

Irrational Management（**非理性管理**）：习惯性的优柔寡断和其他坏管理习惯会导致事实上的决策和慢性开发危机。我们解释了如何利用理性管理决策制定方法来提高项目决心和保证管理人员不脱轨。

Smoke and Mirrors（**虚幻形象**）：演示系统是重要的促销工具，因为它们常常被最终用户看作代表了具有产品质量的能力。急于开拓新业务的管理团队有时会（无意地）鼓励这些误解，做出的承诺超过了机构所能够交付的可运行技术的能力。

Project Mismanagement（**项目管理不善**）：对软件开发过程管理的疏忽会导致失去方向和其他症状。对软件项目的适当监控对于成功的开发活动是必不可少的。产品开发的运行是和建立项目计划一样复杂的活动，而且开发软件就像建造摩天楼一样复杂，同样涉及许多步骤和过程，包括进行检查和权衡。而现实中，关键活动常常被忽视或减少到了最少的量。

Throw It over the Wall（**各管一摊**）：太多的时候，下游管理者和面向对象开发人员会教条地对待本应作为灵活指导原则的面向对象方法、设计模式和实现计划。随着这些指导原则通过认可和公布，它们往往会被认为具有完备性和说明性（但实际上并未达到），并被强制地实现。

212

Fire Drill（**消防演习**）：飞机驾驶员把飞行说成是“数小时的无聊工作然后是15秒的绝对恐怖”。许多软件项目类似于这样的场景：“数个月的无聊工作然后是立即交付的要求。”数个月的无聊工作可能包括拖延的需求分析、再规划、等待资金、等待认可，或者任意数量的非技术技术原因。

The Feud（**管理不和**）：管理者之间的个性冲突会对工作环境产生严重影响。这些管理者的部属常常要承担他们主管之间的争执的不利后果。管理者之间的敌视会在他们的部属的态度和行为中反映出来。

**E-mail Is Dangerous（电子邮件危机）**：电子邮件是软件开发人员的重要交流媒介。不过，
213 对很多主题和敏感的交流来说，它不是合适的媒介。

## 小型反模式：Blowhard Jamboree（吹牛者狂欢）

### 反模式问题

所谓的行业专家的意见常常会影响到技术决策。在大众媒体和非公开出版物上经常会出现批评特定技术的争议性报告。除了承担技术职责，开发人员还要花过多的时间来解答管理者和决策制定者由于这些报告而产生的顾虑。许多这种所谓的专家提供的消息是错误的，有时他们还会提出带有偏见的观点。他们报告的信息常常是二手的。很少有亲身参与的研究和经验来支持他们做出的结论。

### 重构方案

机构内部每项关键技术上的专家对每个开发机构而言都是具有无法衡量的价值的资产。他可以对事实、误传和大众媒体及其他报告中的消息加以区分。如果你的机构中没有内部专家，就指定一些人员来追踪特定技术，通过阅读、培训课程、标准化活动和诸如建立原型等亲身实验来培养他们的专家知识。

电子邮件列表常常会促进误传的扩散。因此，要避免把新闻稿转发到这样的列表上。新闻稿完全可以被称为宣传，它们传播的是经过挑选过对特定机构有益的消息。替代的方法是指示
214 内部专家来分辨重要声明中的事实，然后进行宣讲。

反模式

## 7.3　Analysis Paralysis（分析瘫痪）

**反模式名称**：Analysis Paralysis

**别名**：Waterfall（瀑布）、Process Mismatch（过程失配）

**最常见规模**：系统层

**重构方案名称**：Iterative-Incremental Development（迭代增量式开发）

**重构方案类型**：软件

**根源**：自负、思想狭隘

**不平衡的力量**：复杂性管理

**轶事证据**："我们要重做这个分析，让它更面向对象，使用更多的继承来获得大量的复用。""我们在开始进行任何编码前，先要完成面向对象分析和设计。""嗯，如果用户想根据雇员名字的第4和第5个字母，以及过去4年他们在感恩节和阵亡将士纪念日之间投入时间最多的项目，来建立雇员列表怎么办？""如果你把每个对象属性都当作一个对象，就可以在不相关的类之间复用字段的格式化。"

### 7.3.1　背景

Analysis Paralysis是面向对象开发中的一个经典反模式。面向对象分析强调把问题分解成它的组成部分，但是没有明确的方法来确定系统设计所需详细水平[Webster 1995]。在实际过程中，关注点往往不再是把问题分解到让开发人员易于理解的水平，而是偏移到应用一些方法来达到所谓的"完备性"。而且系统开发人员常常愿意成为Analysis Paralysis的牺牲者，因为"设计从来不会失败，只有实现会失败"。通过延长分析和设计阶段，他们可以避免为结果承担责任的风险。当然这是一个失败的策略，因为在某个特定的时刻之后，总是需要有一个可以工作的实现。

215

### 7.3.2　一般形式

当分析阶段的目标是达到完美和完备时，就会出现Analysis Paralysis。该反模式的特点是推翻和修改模型，以及生成对下游过程没多少用处的详细的模型。

许多新接触面向对象方法的开发人员进行了过多的先期分析和设计。有时，他们使用分析建模作为锻炼自己以熟悉问题领域的方法。但是，他们没有从根本上认识到面向对象方法带来的一个好处就是在领域专家的参与下建立分析模型。如果没有领域专家的参与，当分析过程的目标是建立一个全面的模型时，就很容易陷入困境。

Analysis Paralysis的出现通常是受到了瀑布假设的影响：

- ❑ 在开始编码前可以成功完成详细分析。
- ❑ 问题的所有内容都是预先知道的。
- ❑ 开发时不会扩展或重访分析模型。

面向对象开发不能很好地与瀑布式分析、设计和实现过程相匹配。有效的面向对象开发是一个增量过程，其间会通过设计和实现验证增量的、迭代的分析结果，并把它用作后续系统分析的反馈。

> “啊！人是什么？他来自何方？他又去向何处？”
>
> ——Dan Leno（真名George Galvin，19世纪英国著名喜剧家）

Analysis Paralysis的一个关键指示器就是分析文档对领域专家而言不再有意义。随着Analysis Paralysis的进展，分析模型越发频繁地覆盖领域专家不感兴趣的那些细节。例如，一个医院的健康护理系统的领域模型应该让医院的管理者和职员都能够理解。如果领域模型定义了预期之外的软件概念、分类和特殊内容，分析建模工作很可能已经走得太远了。如果必须向非常熟悉已有系统的人解释这些新类的含义，那么可能对问题已经进行了过度的分析。

216

### 7.3.3　症状和后果

- ❑ 由于人员或项目方向的变化，项目多次重启，对模型进行了很多返工。
- ❑ 在分析阶段不断地反复提出设计和实现问题。
- ❑ 分析成本超过预期，而且没有可以预测的终点。
- ❑ 分析阶段不再涉及与用户的交互，进行的许多分析只是推测得出的。
- ❑ 分析模型的复杂性导致错综复杂的实现，让系统难于开发、文档化和测试。
- ❑ 在设计阶段做出类似于GoF设计模式中使用的那些设计和实现决策。

### 7.3.4　典型原因

- ❑ 项目管理过程设想开发中的各个阶段采取的是瀑布式过程。而实际上，几乎所有的系统都是以增量方式构建的，虽然正式采用的开发过程也许并没有承认这一点。
- ❑ 管理层对他们分析和分解问题的能力比对设计和实现的能力更有信心。
- ❑ 管理层坚持在设计阶段开始前完成所有分析。
- ❑ 没有很好地定义分析阶段的目标。
- ❑ 在分析阶段过程中丧失了规划或领导能力。
- ❑ 管理层不愿做出有关何时对部分领域已经进行了足够定义的牢靠决策。
- ❑ 有关项目目标和交付给客户的内容的前景和焦点出现弥散。分析继续进行下去已经不能提供什么价值了。

### 7.3.5　已知例外

Analysis Paralysis反模式从来都不应该出现例外。

217

### 7.3.6　重构方案

增量式开发是面向对象开发取得成功的关键。与瀑布式开发过程设想对问题存在先验知识不同，增量式开发过程认为在开发进程中才会逐渐认识到问题及其解决方案的细节。在增量式开发中，在每次迭代中都会经历面向对象开发的所有阶段——分析、设计、实现、测试和验证。最初的分析包括对系统的高层次评审，以便让用户对系统的目标和一般性功能进行验证。每个增量都细化了系统中的一个部分。

增量有两种类型：内部的和外部的。内部增量建立对实现的基础结构很重要的软件。例如，可以由第三层数据库和数据访问层构成一个内部增量。内部增量建立多个用例使用的共同基础结构。一般而言，内部增量让重做的工作最小化。而外部增量由用户可见的功能组成。外部增量对通过显示项目的进展来赢得外部舆论非常关键。外部增量对用户确认也很重要。它们通常包括一些用后抛弃的代码来模拟基础结构和后端各层上缺少的部分。选择增量来平衡项目生存、用户确认和成本最小化的影响是项目经理的专有职责。请参阅本章后续内容中的Project Mismanagement反模式来了解针对风险设置增量进度表。

在进行面向对象分析时，往往更容易继续分析而不是结束分析并开始设计软件。由于很多分析方法同样适用于某些层次上的设计，就很容易使用分析阶段来指导整体设计的特定方面。造成这个情况的另一个可能原因是，项目团队会认为设计的同时也需要进行最小限度的分析，因此可以先开始设计再回头进行分析。通常这会导致产品既不像用户可以理解的领域模型，也不像从已实现的系统反推出的设计。

Analysis Paralysis也适用于架构层，其形式与它在开发层的表现类似。对架构的说明毫无疑问可以远远超过让开发人员坚持架构原则和构造时的需要。例如，为设计成仅用于运行在对象基类接口上的架构构造指定已知子类和可允许的子类，就是过分的做法。更合适的做法是在架构构建操作的基类中指定必要的限制，相信（可能还要检验）开发人员在他们的子类中维持了这些限制。

218

在管理层次，Analysis Paralysis被称为微管理，在管理者过度地进行工作分配和对分配下去的任务进行过度的监督时就会发生。

#### 小型反模式：Viewgraph Engineering（视图设计）

**反模式问题**

某些项目中，开发人员会一直在准备视图和文档而不是开发软件。管理层从未获取合适的开发工具，而工程人员没有替代方法，只能使用办公自动化软件来生成伪技术图表和报告。这种

情况让工程人员感到灰心，使得他们真正的才能无法得到发挥，并且会让他们掌握的技能过时。

**重构方案**

许多被束缚在Viewgraph Engineering角色中的开发人员应该被重新分配去构造原型。许多机构没有充分使用构造原型这一方法。该方法在项目中除了建立销售工具，还可以扮演很多其他重要角色。原型可以回答无法从报告分析中演绎出答案的技术问题，还可以用于减少许多类型的风险，包括与技术和用户接受性有关的风险。构造原型有助于缩短对新技术的学习曲线。

原型主要可以分为两类：模型和工程原型。模型（mockup）是模拟用户界面外观和行为的原型。当把模型和系统可用性试验（以及实际最终用户）结合到一起时，就可以对需求事项和用户接受性进行评估。工程原型（engineering prototype）包括一些运行功能，例如应用服务、数据库和与遗留系统的集成。在产品系统开发前使用替代语言（例如Smalltalk）建立系统的原型是有益的练习。这个原型有助于在使用更有效、但是成本更高也更缺乏灵活性的技术之前对架构进行验证。

219

反模式

## 7.4　Death By Planning（规划致死）

**反模式名称**：Death by Planning
**别名**：Glass Case plan、Detailitis Plan
**最常见规模**：企业层
**重构方案名称**：Rational Planning（理性规划）
**重构方案类型**：过程
**根源**：贪婪、无知、匆忙
**不平衡的力量**：复杂性管理
**轶事证据**："我们在制定完整的程序计划前不能启动。""计划是惟一能够保证我们成功的东西。""只要我们遵循计划不偏离它，就会取得成功。""我们有一个计划，我们只需要遵循它！"

### 7.4.1　背景

在许多企业的文化中，详细的规划是所有项目都设定要进行的活动。这一设定对制造性活动和许多其他类型的项目是合适的，但对许多软件项目则不一定合适，因为它们在本质上就包含许多未知的和无序的活动。在过于认真地为软件项目进行详细规划时，就会出现Death by Planning反模式。

### 7.4.2　一般形式

许多项目由于过规划而失败。过规划常常是成本跟踪和人员利用率监控造成的后果。过规划 221
的两种类型分别是Glass Case Plan和Detailitis Plan。Glass Case Plan是Detailitis Plan的子集，在项目启动时会停止（过）规划。而在Detailitis Plan中，过规划会由于一些无法满足的原因而一直延续到项目不再存在。

#### Glass Case Plan

在项目开始时建立的计划即使从来没有被更新过，也常常会被当做是该项目准确的、当前的视图。这种做法可以在项目启动之前就给管理层一个有关交付的"舒服的视图"。可是，如果不再对计划的执行进行追踪，或者不再更新它，随着项目的进展它会越来越不准确。与这个虚假的视图一起，常常还伴随着缺乏有关项目进度的具体信息，而往往只有在某次关键交付错过了进度安排时才会让人意识到这一点。

图7-1显示了一个在项目启动前建立的项目计划。管理层设想计划可以自动保证交付——完全按照指定的时间而无需任何干预（或者管理）。

#### Detailitis Plan

有时，用于有效交付的解决方案被看成是通过连续规划活动实施的高度控制，而这些规划活 222

动涉及大部分高级开发人员以及管理者。这种方法常常会发展成层次化的一系列计划，显示出额外的(也是不必要的)细节。定义这样高水平的细节的能力让人觉得项目是完全受控的(见图7-2)。


图7-1　Glass Case Plan

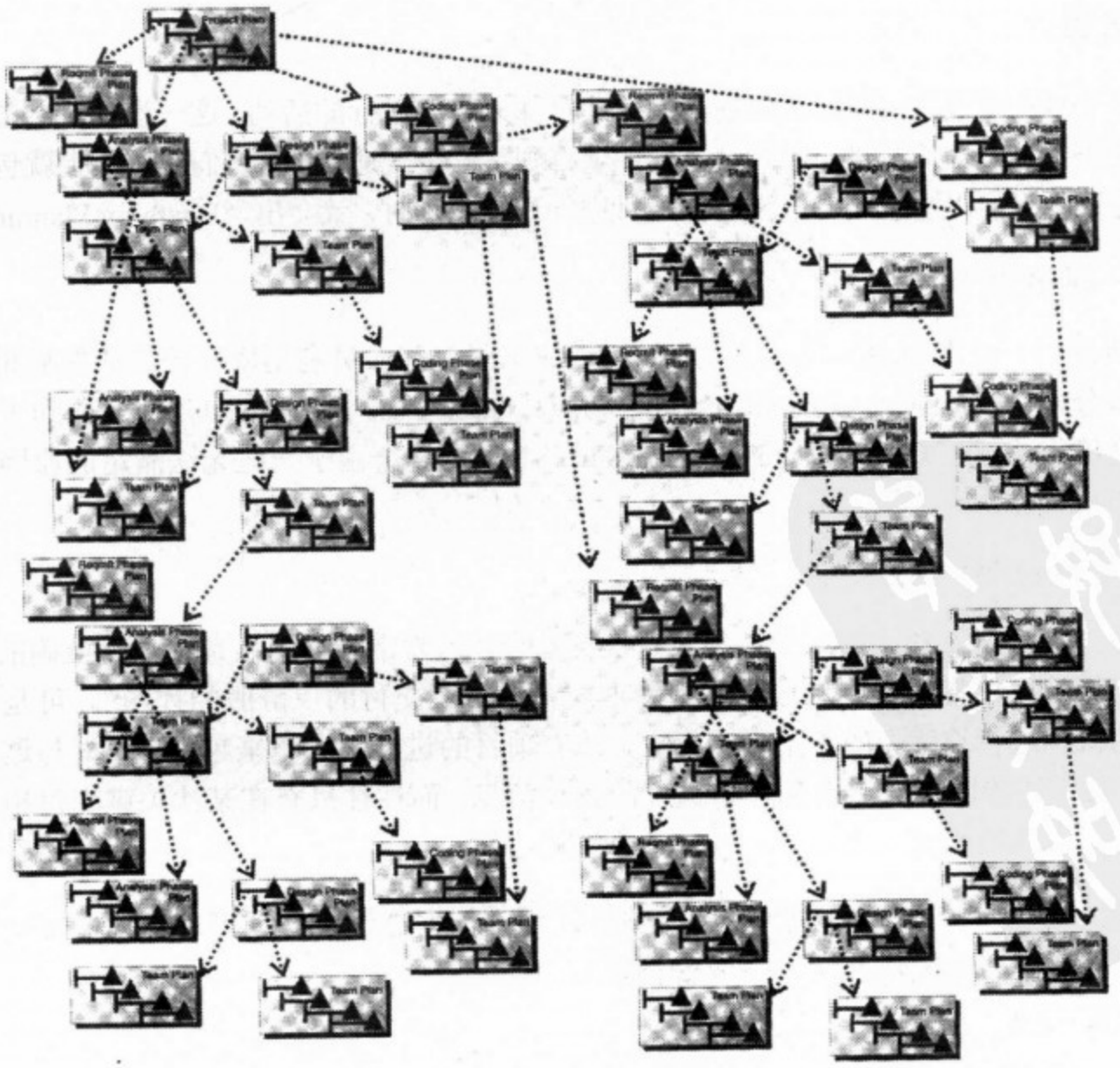


图7-2　Death by Planning要求彻底的先期规划，而不管是否存在未知因素

### 7.4.3　症状和后果

在Glass Case Plan和Detailitis Plan的症状中，首先发现的是Glass Case Plan的症状。

#### Glass Case Plan

它通常至少包括下面的一种症状：

- 无法在注重实效的层次进行规划。
- 关注成本而不是交付。
- 只要项目有资金，就贪心地想挖掘所有的细节。

它的后果是增量形式的：

- 不了解项目开发的状态。制定的计划没有意义，而对交付的控制随着时间进程逐渐减少。项目可能远远超过期望的交付状态，也可能远远落后，但是没有人知道实际情况。
- 未能交付关键的交付品（最终后果）。

它带来的后果逐渐恶化直到项目最终（在时间和成本上）超出计划。这时通常可以选择的处理方法有：

- 进一步的投资。
- 危机项目管理。
- 取消项目。
- 解雇关键人员。

> “为了艺术而追求艺术。”
>
> ——Howard Dietz（美国著名作家）

#### Detailitis Plan

它的症状是Glass Case Plan症状的扩展集：

- 无法在注重实效的层次进行规划。
- 强调成本而不是交付。
- 在规划、描述进程的细节和再规划上而不是交付软件上花过多的时间：
  - 项目经理规划项目的活动。
  - 项目领导规划团队和开发人员的活动。
- 项目开发人员把他们的工作分解成更细的任务。

它的后果纠缠在一起，相互支持：

- 每个规划人员必须按照他的计划中显示的层级来监控和捕捉进度并进行重估。
- 无休止的规划和重规划导致进一步的规划和重规划。

223
224

- 目标从交付软件偏移到交付一组计划。管理层认为由于对工作和成本进行了跟踪，取得的进度一定是与投入等价的。实际上，进度和投入之间没有直接的关系。
- 软件交付连续延误，项目最终失败。

### 7.4.4　典型原因

Glass Case Plan和Detailitis Plan两者的首要原因都是在规划、制定进度表和捕获进度方面缺乏注重实效的合理方法。

#### Glass Case Plan

- 没有符合当前情况，显示出软件构件交付品及其交付日期的项目计划。
- 忽视基本的项目管理原则。
- 过分热心的最初计划，试图对开发实施绝对控制。
- 为了获取合同进行的销售辅助手段。

#### Detailitis Plan

- 过于热心的连续规划，试图对开发实施绝对控制。
- 规划成为首要的项目活动。
- 强迫客户服从计划。
- 强迫行政管理层服从计划。

### 7.4.5　已知例外

225　Death by Planning反模式从来都不应该出现例外。

### 7.4.6　重构方案

Glass Case Plan和Detailitis Plan反模式的解决方案是一样的。项目计划应该显示出主要的交付品（无论项目有多少个团队）。应该在两个层次上确定交付品：

（1）产品。就是卖给客户的制品，客户包括公司内部相关人员。
（2）（产品中的）构件。支持业务服务所需的基本技术制品。

交付品包括：

- 业务需求声明。
- 技术说明。
- 可测度的接受性准则。
- 产品使用场景。
- 构件用例。

应该使用确认里程碑来补充每个构件和整个产品。这些里程碑包括：

- 概念设计认可。
- 说明设计认可。
- 实现设计认可。
- 测试计划认可。

应每周更新交付品计划，以保证通过适度的规划和控制来减少项目风险。这样才能通过适当的、及时的响应来解决各种问题、风险以及规定的交付品的延迟或提前交付。

如图7-3所示，要根据估计的完成度进行跟踪。有时这意味着逆转以前某个时间输入到计划中的完成度。完成度应该是总体粗略的度量而不是精细的度量。例如，使用25%而不是5%作为度量的幅度。

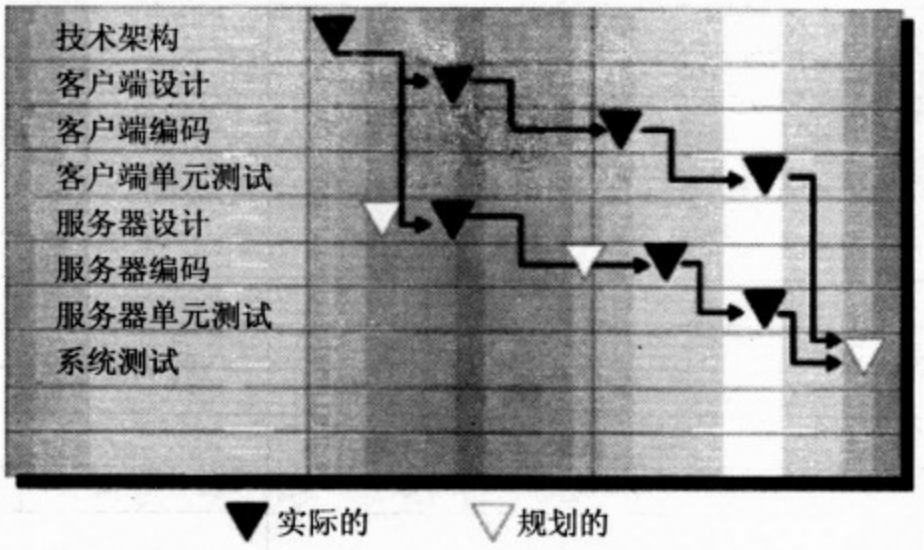



图7-3　正确的规划涉及对实际进度的跟踪和重规划

可以有效地使用甘特图来可视化显示交付品及相应的日期和它们之间的相互依赖。通过针对基线计划进行跟踪，可以立即看到交付品的下列状态： 226

- 按期。
- 已交付。
- 提前（以及估计的新交付日期）。
- 延误（以及估计的新交付日期）。

重要的是尽早制定基线，而且尽量少改动它。否则就会丧失对变化进行跟踪的能力。如果进一步的活动/任务/交付品之间存在依赖，就在计划中设置这些依赖。

进行估计时，明智的做法是为所有那些不可避免的“未知”问题留出一段附加时间。这些“未知”问题可能有：

- 需求增长（蔓延）。

- 设计僵局。
- 绕过第三方软件。
- 确定缺陷（在一组集成的构件中找到问题）。
- 纠正缺陷（除错）。

为完成任何活动所需的时间建立最小期限同样重要。这样可以防止类似于要在两天内编码和测试一个“简单”程序的强制要求。

227

### 7.4.7　变化

Death by Planning的变化是在细节水平上的不同，可以涉及从通常与投资/认可阶段联系在一起的主要里程碑的确定，到项目交付阶段每个团队的微型交付品（见图7-4）。

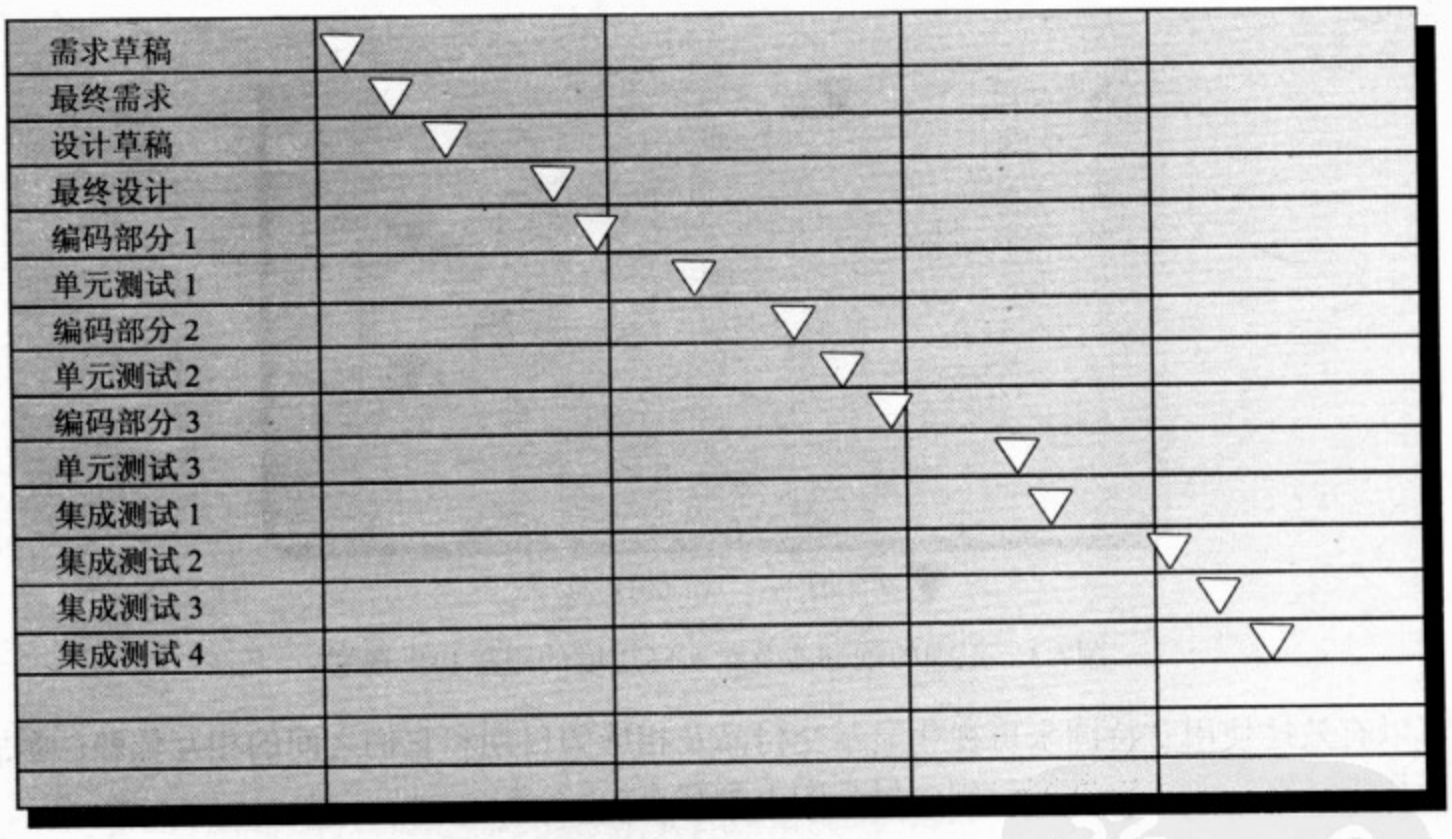



图7-4　投资里程碑为项目的继续建立决策点

下面这些变化同时适用于Glass Case Plan和Detailitis Plan：

- 投资变化（见图7-4）。
- 微型交付品变化（见图7-5）。

微型交付品计划的Glass Case Plan版本与Detailitis Plan版本的区别只是从来不会对它进行更新。它显示了在项目开始之前无法完全理解的非常微小的交付品。因此，如果对要构建的软件缺乏任何的真实理解，在此基础上建立的估计从定义上来说就都是不正确的。这类计划通常是有技术天分的业余人士建立的。虽然要清晰地理解任务，但是把它们放在计划中只会导致工作中不必要的规划和跟踪（在Detailitis Plan中）活动。

228

| 需求 | | | |
|---|---|---|---|
| | 需求草稿 | | |
| | | 客户端GUI | |
| | | | 范围 |
| | | | 功能 |
| | | | 约束 |
| | | | 使用场景 |
| | | | 性能标准 |
| | | 客户端应用 | |
| | | | 范围 |
| | | | 功能 |
| | | | 约束 |
| | | | 使用场景 |
| | | | 性能标准 |
| | | 应用服务器 | |
| | | | 范围 |
| | | | 功能 |
| | | | 约束 |
| | | | 使用场景 |
| | | | 性能标准 |
| | | 安全服务器 | |
| | | | 范围 |
| | | | 功能 |
| | | | 约束 |
| | | | 使用场景 |
| | | | 性能标准 |
| | 最终需求 | | |
| | | 客户端GUI | |
| | | | 范围 |
| | | | 功能 |
| | | | 约束 |
| | | | 使用场景 |
| | | | 性能标准 |
| | | | |
| | | | |

图7-5　微型交付品计划

## 7.4.8　示例

这些例子是我们的亲身教训。

### Glass Case Plan

在这个例子中，我们会看到一个系统集成商决定构建一个虽然一年多以前就发布了国际标准，
230 但仍无法从任何主要供应商那里获得的中间件构件。系统集成商同意在开始所有项目工作前先提供一份详细交付计划以获取投资。该计划的基础是没有“恼怒地”交付过软件的人员所作的估计。计划在技术上非常详细，做出的计划也非常乐观。项目管理人员频繁地引用它，但从未更新它来反映任何实际工作。这导致错过了交付日期。总体上缺乏对项目实际进度的了解，在已经过了交付日期之后才通知系统集成商不能按期交付。图7-6中显示的就是这样一个开发项目计划的节选。

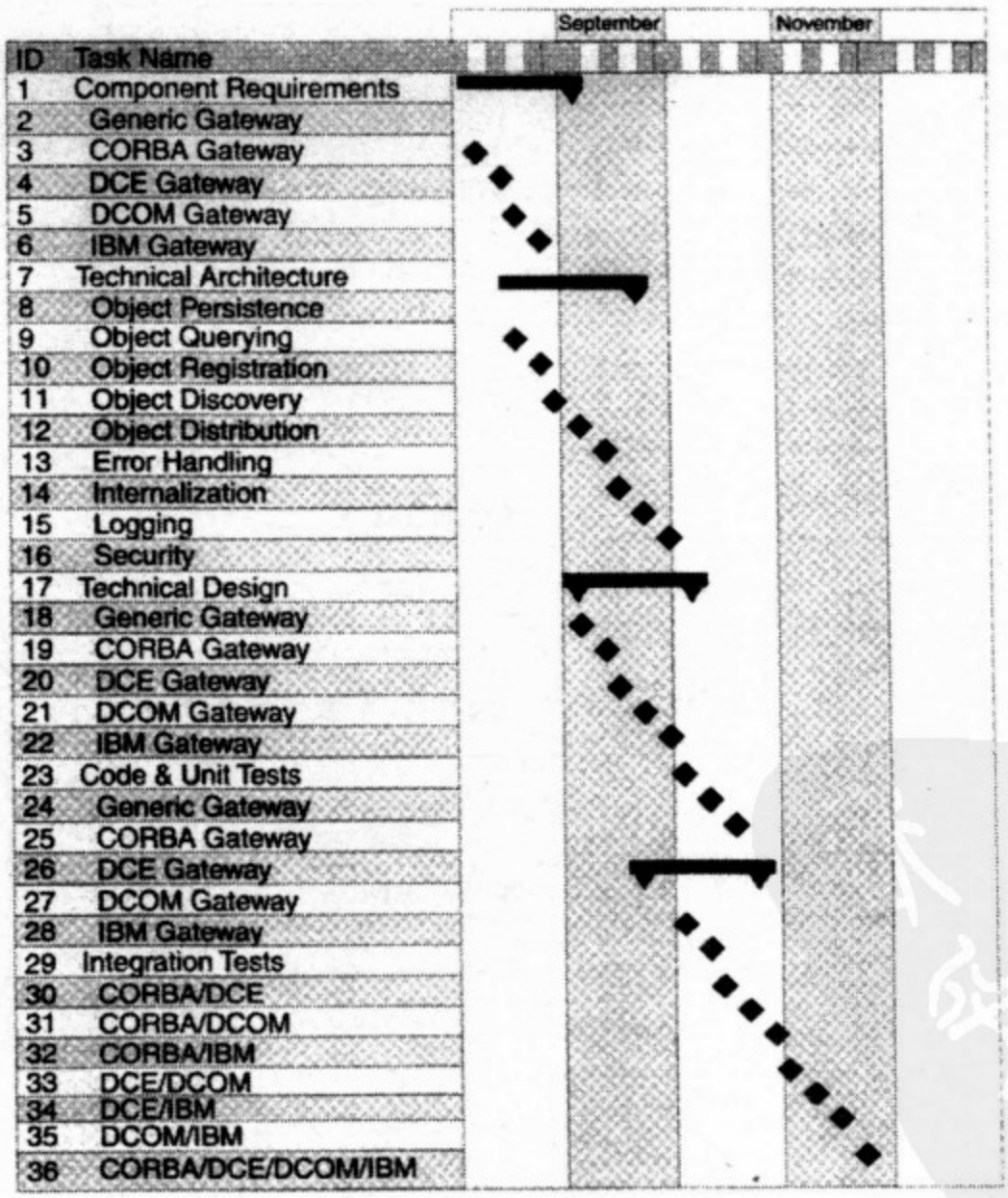



图7-6　Glass Case Plan示例

### Detailitis Plan

在控制开发以保证建立完全控制的尝试中，最终用户公司提出了一个有三个层次的计划：

（1）开发阶段。
（2）团队的任务和工作。
（3）团队成员的任务和工作。

图7-7中的计划显示了它的复杂性。

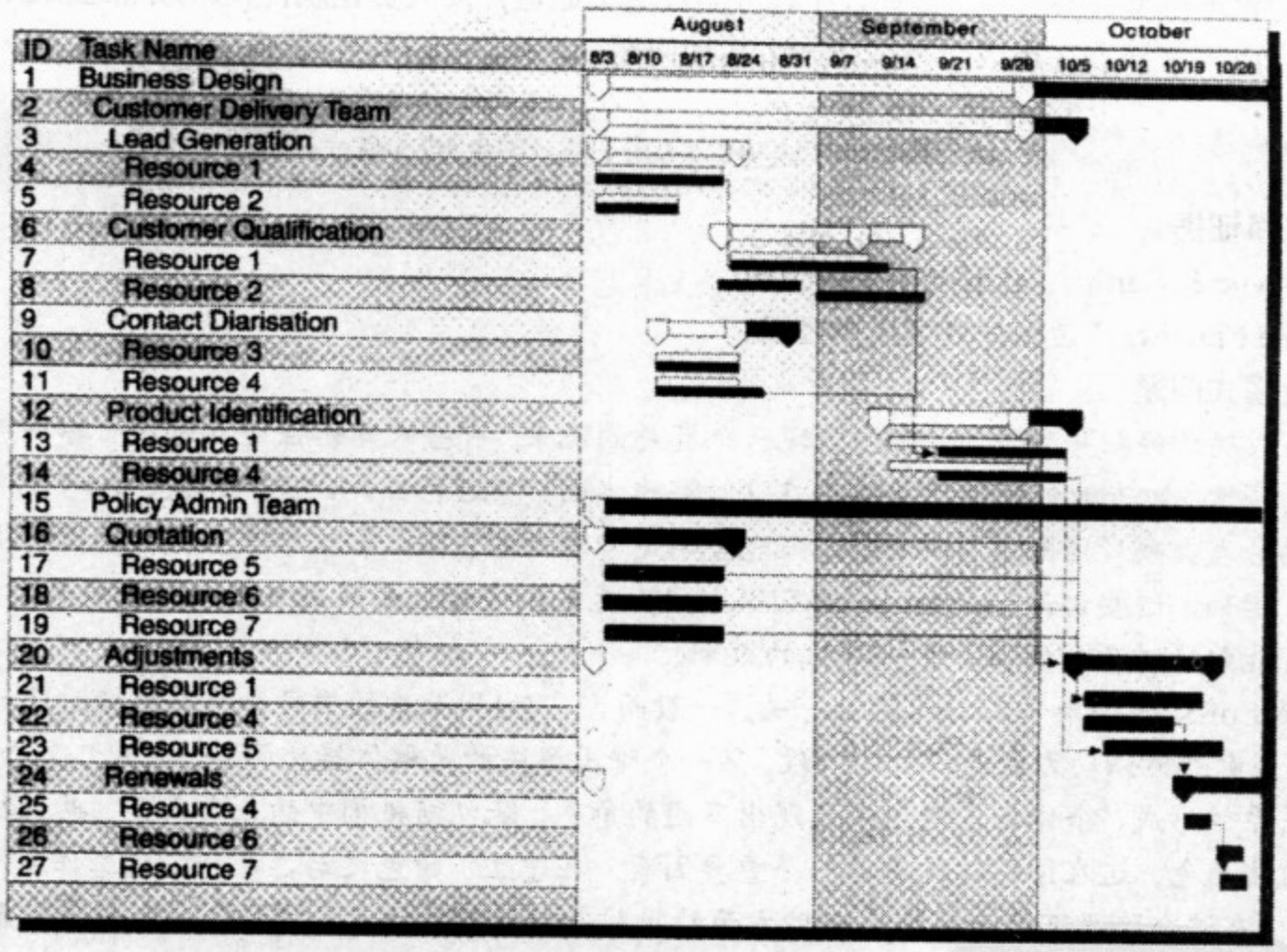



图7-7　Detailitis Plan示例

Detailitis导致如果不将相当的注意力从交付系统的工作上移开，就不能针对计划进行跟踪。其后果是人员效率的显著降低。由于复杂性的原因，计划管理迅速变得不切实际。

解决方案是用显示关键交付品及其交付日期，并包括它们的相互依赖和约束的计划来替代详细计划。如果错过了某个交付品，那么一定发生了某些重要的事情。原先的计划试图根据不切实际的工作量来跟踪进度：工作量＝人员数量×天数。那又怎么样？$E=MC^2$，但是公式本身并不会制造核能！

## 7.4.9　相关解决方案

Analysis Paralysis反模式会加剧Death by Planning反模式的后果。如果分析阶段耗费的时间超过了进度表上分配给它的时间，那么要么必定会错过某个进度，要么就会在后续阶段中使用不全面的分析模型。

### 7.4.10　对其他视角和规模的适用性

Death by Planning与软件开发的混乱（或称白浪，即河中的湍流）本质并不匹配，因为它在管理层的规划模型和实际开发活动之间造成了显著的不同。架构师和开发人员常常要过双重生活：一方面，他们显然必须与管理层的项目计划合作；同时，他们还要面对软件的实际状态，而它可能与管理模型并不一致。例如，计划可能被用于逼迫开发人员在软件模块成熟之前就宣布它
232 们已经完成。这导致在集成和测试中产生后续问题。

#### 小型反模式：Fear of Success（成功恐惧症）

**轶事证据**

Wayne & Garth: “我们配不上!”

The Crazies: “老兄，这是成功恐惧!”

**反模式问题**

当成功已经触手可及时，常会出现一个有趣的现象。有些人开始强迫性地担心各种可能会出错的事情。对专业能力的不安全感在这时浮出水面。在进行公开讨论的时候，这些担忧和不安全感会占据项目团队成员的思维。可能会制定非理性的决策，采取非理性的行动来解决这些顾虑。例如，这些讨论可能会在项目团队之外产生负面宣传，影响到其他人对交付品的看法，甚至可能最终对项目结果产生破坏性的效果。

Fear of Success和项目终止问题有关。一般而言，在1周长度的项目和更长持续时间的工作中，都会看到群体动力会经历一些阶段。第一个阶段解决的是群体接受问题。在第二个阶段中，随着关系的形成，个体会在群体中呈现出不同的角色，既包括机构中的正式角色，也包括非正式的自主角色。这在团队建设中是一个重要因素。在建立了角色之后，才会完成工作（第三个阶段）。在这个阶段可能出现许多个性方面的问题。由于项目的完成可能会导致群体的解体，这些问题常常会在项目接近终止的时候出现（第四个阶段）。在终止阶段，常常会以间接的方式来表达对项目结果、它将来的生命周期、以及群体的后续活动等方面的顾虑。也就是说，大家会做一些疯狂的事情。

**重构方案**

233 管理层在接近项目结束时可以采用的一项重要行动是宣布成功。即使实际结果可能不太明确，也需要管理层做出声明来为项目的成功结果提供支持。重要的是帮助项目团队接受他们的成就的重要性和他们认识到的教训。宣布成功还有助于减轻终止问题，维持团队对机构的投入，并为将来的项目活动做好铺垫。举行庆功仪式来授予证书或奖牌，是对这一意图的适当表示。表达对专业能力的认可是很廉价的，但是受到接受者的高度重视。

**变化**

个人咨询和指导是解决个体顾虑的有效方式。有多个项目经验的高级人员可以帮助其他人解决终止问题带来的压力。在向成功过渡的过程中体现出耐心是应该表现出来的行为，并且应
234 该受到效仿。

反模式

## 7.5　Corncob（玉米棒子）

**反模式名称**：Corncob

**别名**：Corporate Shark（公司鲨鱼）、Loose Cannon（炮筒子）、Third-World Information Systems Troubles （TWIT）

**最常见规模**：企业层

**重构方案名称**：Corncob Removal Service

**重构方案类型**：角色

**根源**：贪婪、自负、思想狭隘

**不平衡的力量**：资源管理、技术转移管理

**轶事证据**："为什么Bill这么难于共事？""管理层总是听取叫得最久，声音最大的人的意见。""我对开发负责，所以你们必须服从我做出的决定。""我们要改变构建过程，并且接受我们要晚1到2个月交付的情况。"

### 7.5.1　背景

Corncob就是在软件开发工作中很常见的难于相处的人。这种态度可能是由于个性方面的原因，但更常见的是由于个人出于获得认可或金钱的动机而造成的困境。

由于严格的进度表和预算的压力，软件开发会变得令人紧张。Corncob会加剧这些问题，在可能已经紧张过度的环境中产生不必要的额外压力。 235

"Corncob"是软件联盟OMG常用来描述难于相处的人的行话。

### 7.5.2　一般形式

难于相处的人（Corncob）以破坏性的行为在软件开发团队中造成问题，更糟的是甚至可能是在企业中造成问题。这样的人可能是团队的成员，也可能是外部高级职员中以不同的方式对团队施加不利的技术、非技术或个人影响的人（例如技术架构师或开发经理）。和Corncob打交道时，重要的是记住非技术就是行使权力，而Corncob更关注非技术而不是技术。他们通常是在个人和/或组织层次上操纵非技术的专家。关注技术的人会在不情愿的情况下很容易地成为Corncob的策略对付的牺牲品。

### 7.5.3　症状和后果

- 由于某个人不赞成开发团队或项目的关键目标或基本过程并不停地试图改变它们，团队或项目无法取得进展。

- 某个人在担忧的伪装下不停提出难以处理的反对意见：性能、可靠性、技术市场份额，等等。
- 企业中的许多人都清楚地知道这些破坏性行为，但是管理层（以某种方式）容忍和支持这种行为，因为他们不了解，或不愿意解决这种行为造成的损害。
- 非技术力量形成了很难让技术讨论不脱离正轨的环境。
- 非技术力量导致系统的范围或需求频繁改变。项目变得比“先发制人”还容易起反作用，因为每个人都在对Corncob的“无止境的改进”做出反应。

236
- 具有破坏性的人常常是未受高级软件开发经理或项目经理直接指导的管理人员。
- 公司没有确定的决策过程来解决问题。这让管理人员可以不适当地干涉超出他的职权范围的事。

### 7.5.4　典型原因

- 管理层不承认Corncob的行动造成了影响，也就助长了Corncob的破坏性行为。管理层对情况的看法往往是由Corncob来提供的。
- Corncob有一个隐藏的日程表，它与团队的目标相冲突。
- 团队成员之间存在根本性的争论，任何数量的交流都无法解决。
- 管理层没有把开发群组与内部和外部的影响力量隔离开，不适当地分配了人员的角色，让他们可以为了自己的目的滥用权力。更糟的是，管理层甚至可能根本没有对责任进行分配。

### 7.5.5　已知例外

当公司或产品开发经理愿意忍受Corncob的行为时，Corncob反模式是可以接受的。这取决于对效益和缺点的主观判断。

在涉及多个机构的项目中，指定一个Corncob来扮演已有架构的保卫者来避免不适当的变化，在某些时候是有益的。当有多个存在冲突的技术观点时，常常需要具有支配力的个性才能实施参考架构。

### 7.5.6　重构方案

对Corncob反模式的解决方案可以在管理职权的多个层次上加以应用，包括战略层次、战役层次和战术层次。在所有情况下，对破坏性行为的管理层支持是关键的行动杠杆。通过消除上述

237 层次上的管理层支持，Corncob会失去支撑，而软件开发团队最重要的利益可以占据支配地位。

#### 战术解决方案

战术解决方案在工作过程中，例如会议中即时采取的行动。它们包括下列行动：

- 转移责任。让反对者承担在大家都同意的时间内解决他们确定的问题的全部职责。也就

是说，把规划和解决的责任交给提出这些顾虑的人。

- 隔离问题。Corncob的观点往往是孤立的。引起群体的关注是抵御个人行为和孤立考虑的有效方式。如果Corncob采取了对抗的态度，要记住永远不要个人承担任何事，那样做总是错误的。集合群组来讨论和限制住关键的反对意见，然后尽量通过大多数人的意见来否决反对者。民意测验是展现群体意见的有效方法。
- 质疑问题。当Corncob使用模糊的或“意味深长的”词语或表达时，应要求他澄清他的意思。当他使用道听途说的证据来做出断言时，就要求他证实他的主张并确定其立场。

**战役解决方案**

战役行动是在有限的组织结构范围内离线进行的。主要包括：

- 矫正性会谈。管理层单独会见导致问题的人，并解释他的行为造成的影响。矫正性会谈的目的是唤起对方对后果的警觉，并就如何改变其行为达成一致。
- 友好地介绍新职。建议Corncob利用猎头公司帮助他体面地走出困境。

**战略解决方案**

战略行动是长期的，并且是在更广泛的企业范围内进行。例如：

- Corncob支持组。如果在机构中有多个Corncob，管理层可以把他们调到同一个群组中。这样，他们就必须互相竞争。常见的是，他们中的每个人都会去找管理层，询问为什么要让他和如此难于相处的人共事。管理者抓住机会说明对方本身在其他人眼中同样是难于相处的。这会让Corncob对自己的行为自省，并可能得以改进。 238
- 空部门。难于相处的管理人员可能被调往只有他一个员工的部门。这些管理人员通常会提前申请退休或去别的公司寻求工作。
- 强制削除。有时没有别的办法，只能把难于相处的人从项目团队或机构中开除掉。

### 7.5.7　变化

前述解决方案的变化包括：

- Sidelining（边缘化）。这一做法是重新分配Corncob只承担最小限度的职责。我们发现Sidelining是不太有效的实践，因为它让Corncob有更多的时间来考虑和暗中运作他自己的（往往是隐藏的）目标。
- Third-World Information System Troubles（TWIT）。这个术语是由医学博士Randall Oakes提出的，指的是在当前环境下都会受到负面后果影响的情况下却依旧抵制IT变化的人。TWIT在某种程度上从现状中受益。就像Oakes博士指出的一样，帮助别人处理压力问题可以促进信任、友谊和更为开放的思想。
- Corporate Shark（公司鲨鱼）。Corporate Shark是有经验的管理者，主要是关系管理而不是

专业技术，后者甚至可能由于前者的原因而长期被忽视。他们的生存诀窍是他们认识的人而不是他们知道的事。Corporate Shark知道如何“运用制度”，常常给那些专注于技术的人建立复杂的工作环境。在工作中，每个人都需要有朋友。最好的朋友就是在公司中有影响力，并在Corporate Shark攻击你时能拔刀相助的人。管理与Corporate Shark的关系的最佳方式是完全避开他们，因为和他们在一起的惟一后果只会是负面的。

- Bonus Monster。有些人喜欢在软件开发进程中走捷径来获得短期的利益（例如行政部门的奖励）。Bonus Monster尤其具有破坏性，因为他们鼓励在企业中不同部门之间进行内部竞争。通过改造或取消奖励体系可以纠正这个问题。Bonus Monster往往也是Corporate Shark，不过是具有强烈动机要获得可以产生奖励结果的鲨鱼。

239

- Firebrand（纵火者）。Firebrand就是故意制造非技术突发事件的人。例如，他可能会制造各种各样的错误指示和障碍来使某部分关键软件的延期交付。他的目的是制造突发事件，以便让他自己以后可以成为拯救整个机构的英雄。为了取得成功，Firebrand会灭掉火。也就是说，他会根据需要排除所有障碍，重新调整开发人员的注意力，让他们取得成功。
- Egomaniac（自大狂）。Egomaniac总是把自己想像成关键影响者或支配人物，受到这种想像的困扰。他们也称为Prima Donnas（首席女歌手）。对于成为了Egomaniac的高级管理人员，补救方法就是把他们调往一个空部门。在会议中，可以公开认可Egomaniac的专家经验和重要性来对他们加以管理。不过，有时这种做法会事与愿违，可能会进一步促进他们的自我陶醉倾向。
- Loose Cannon（炮筒子）。某些人因为其外向的性格而难于相处。他们的行为会对项目形象和士气产生破坏性的影响，例如不停地透露信息和对重要组织关系不敏感。在群体环境中，他们很快就会让别人意识到他们的存在。解决Loose Cannon的最有效方法是通知管理层应该进行矫正性会谈。
- Technology Bigot（技术顽固者）。Technology Bigot就是传播用于推销的欺骗信息，拒绝考虑其他观点的Loose Cannon。例如，那些坚持所有信息技术都应该是基于微软产品的人，或者那些坚持CORBA是实现分布式对象的惟一正确方法的人，都属于Technology Bigot。

“仅取得成功并不够。其他人必须失败。”

——Gore Vidal（美国著名作家）

- Territorial Corncob（领土玉米棒）。试图保护自己的组织或技术势力范围的人常会表现出防卫性的行为。他们要保卫自己的领地以减轻暗藏着的对自己能力缺乏信任的不安全感。Territorial Corncob喜欢受到恭维，而且对自己的弱点很敏感。要避免激起他们的贪念。一旦Territorial Corncob感觉到在他们的领域范围中潜在的新机会或权力，就会牢牢抓住。

240

- Corncobs out of the Woodwork。在出现了特别困难或紧张的情况，例如严重的项目问题或面临解雇时，某些Corncob会加强他们的行动。在面临解雇时，如果你有足够的能力获得另一项工作，可以考虑自愿离职让其他人可以保留他们可能迫切需要的工作和薪水。

- Saboteur（破坏者）。Saboteur是那些将要退出群体，并开始为了他们以后的工作而操纵当前工作环境的人。例如，Saboteur常会试图通过积极的征募、谣言和负面行为来让同事离开项目。这种情况尤其难以发现和补救，因为Saboteur常常会隐藏自己离开的意图。有一家大型的计算机制造商经常举行提供大量饮酒的业余派对，鼓励雇员们在派对上讨论在公司外的职业计划。Saboteur常常会有几个知道他的计划的知己。一旦发现了Saboteur，就要阻止他对工作环境造成影响。
- Careerist（野心家）。他是Saboteur的一种变化，在他的影响下做出的技术选择可以扩展他个人的经验和工作的适应性。例如，一名技术架构师可能会选择使用对象数据库来保持对象持久性，虽然一个具有对象API的关系数据库才是更为合适的技术方案。
- Anachronist（落伍者）。这是由于他不能理解就拒绝创新的Corncob。Anachronist对遗留技术可能很有见地，所以能武断地拒绝新技术。对他们要进行教育培训。

### 7.5.8　示例

几年前我们遇到的一个Corporate Shark最近被调到一个单人部门，然后被解雇了，因为他的个性造成的负面后果超过了他能够给公司带来的效益。根据我们的经验，Corncob最终会自我毁灭，离开公司。不利方面是他们有时还会回来。

我们的一名朋友在退出公司的奖励体系时性格发生了巨大变化（朝向好的方面）。在当时 241
的奖励体系下，这位朋友想获得短期结果（行政奖励）而参与了对多个软件开发进程的对抗性干预。

Territorial Corncob可能是最粗暴的变化形式。尤其是在某个人或某群人相信某个特定技术术语，如“架构”“属于”他们时，他们可能会试图猛烈地压制任何使用这个词的其他人。电子邮件是Territorial Corncobs最有效的武器，因为它是最确凿的证据。电子邮件还是在对抗和争论中易于使用的媒介。

### 7.5.9　相关解决方案

本反模式的Technology Bigot变化也是Golden Hammer反模式的一种形式。参阅Golden Hammer反模式以获得相关的解决方案。

### 7.5.10　对其他视角和规模的适用性

Corncob对架构和开发的影响通常是减慢进程。Corncob产生的对抗性问题会让讨论的广泛程度降低，转而只关注特定的影响力量。某些时候，Corncob行为会反转设计模式意图带来的效益。Corncob不会让软件过程更广泛深入和更有效，而是正好相反。Corncob倾向于让讨论和决策的制定偏向于他选定的、但也许并不是最重要的影响力量，而不是通过考虑所有显著作用力来让开发过程更为平衡。 242

## 小型反模式：Intellectual Violence（智力暴行）

### 背景

微积分是关于函数和变量替换的数学原理的直观理论。它是LISP编程语言的内在重要理论之一。尽管只有部分大学向学生教授这门计算机科学课程，但是经过这门课程培训的人常常会假设所有人都了解微积分。这种误解可能就是Intellectual Violence的实例。简短的非正式教学就可以清除所有的误解。

### 反模式问题

当某些理解了某种理论、技术和术语的人在会议中使用这些知识来胁迫他人时，就是出现了Intellectual Violence。由于技术人员通常不会表露出他们不了解的地方，这个反模式可能是在不经意间发生的。简而言之，Intellectual Violence是交流的失败。当项目中的部分人或大多数人不能理解某个新概念时，由于他们要掩饰他们的自卑感或完全回避该话题，项目进度可能会无止境地停滞不前。如果Intellectual Violence普遍存在，就会形成一种抑制了生产率的防卫性文化。大家会控制和隐藏信息而不是共享信息。

### 重构方案

有一种以指导为基础的组织文化可以作为替代。在这种文化中，大家经常进行交叉培训。应该认识到无论每个人在机构层次上的位置如何，他们都有独特的才能和知识。在指导文化中，大家受到鼓励分享他们的知识来促进机构的整体成就。领导以身作则可以最好地推动指导文化。如果机构中的负责人主动参与指导活动，在机构的所有层次上都会更广泛地接受这一文化。鼓励大家共享信息是利用知识资源、限制重新发明的有效手段。

反模式

## 7.6　Irrational Management（非理性管理）

**反模式名称**：Irrational Management

**别名**：Pathological Supervisor（病态主管）、Short-Term Thinking（短期思维）、Managing by Reaction（反应管理）、Decision Phobia（决策恐惧症）、Managers Playing with Technical Toys（玩技术玩具的管理者）

**最常见规模**：企业层

**成功方案名称**：Rational Decision Making（理性决策）

**成功方案类型**：角色和过程

**根源**：责任（通用原因）

**不平衡的力量**：资源管理

**轶事证据**："谁负责这个项目？""我希望他下定该死的决心了！""我们现在做什么？""我们最好在启动前和管理层说清楚。""不要去问，他们只会说不。"

### 7.6.1　背景

Irrational Management覆盖了软件项目中的不少常见问题，这些问题都可以追溯到负责该项目的人的个性上。例如，管理者可能对某些技术方面具有强迫性的兴趣，或者存在个性限制，导致他们成为低效的或非理性的管理者。可以把Irrational Management看作一组不太合适的优先级。这时，尽管管理者清楚自己确定的优先级非常不合理，但是仍然用它们把软件项目引导到非理性的方向。

245

### 7.6.2　一般形式

一个或多个开发项目的管理者（或管理团队）无法制定决策。这可能是由于个性缺陷，或者是受细节困扰的结果。这些细节可能是管理者的个人兴趣或行为，例如技术"玩具"或者微管理。面对危机时，管理者的决策是"膝跳反应"而不是经过深思熟虑的战略或战术行动。这些反应常常导致更进一步的问题。优柔寡断和快速反应构成的循环越来越频繁，其后果的严重性也会逐渐升高。

Irrational Management反模式会由于管理者不能指导开发部属而显著加剧。这也被称为缺乏良好的人员管理技能，其特点是无法完成以下活动：

- 认识到部属的能力，既包括他们的强项，也包括他们的弱点。
- 提供适合于部属技能的清晰目标。
- 与部属有效地交流。

### 7.6.3　症状和后果

Irrational Management的首要症状是项目拷问，也就是对某个关键话题进行争论。必须做出决策来让开发部属继续前进。项目拷问有多种后果。

- 部属的挫折感增加。
- 交付的延期增加。
- Corncob可以利用这种环境。

### 7.6.4　典型原因

管理者缺乏对下列对象的管理能力：

- 开发部属。
- 其他管理人员。
- 开发过程。

他还缺乏清晰的愿景和策略，因此：

- 无法制定决策。
- 害怕成功（参阅相应的小型反模式）。
- 不知道项目活动和交付品的真实状态。

246

### 7.6.5　已知例外

Irrational Management反模式不应该有任何例外。不过，在规划更好的长期解决方案时，可以在一定程度上容忍一个具有大师级技能，在某些方面完全独一无二并且可以提供主要技术优势的“金童”的存在。

### 7.6.6　重构方案

按照下面的指导原则来解决Irrational Management反模式：

（1）承认你存在问题并寻求帮助。当管理者受到前述典型原因困扰时，他们首先必须承认自己存在问题。假定他们不清楚自己的情况，第一个步骤就是确定关键的指示器，最常见的方式就是“付出代价”。例如，非理性管理者在问题变成了危机时才会发现它，而不是由技术部属对正在增长的问题加以解决并寻求帮助。管理者必须使有才能的部属（或顾问）聚集一起并愿意听取他们的意见，让他们分担复杂的管理工作。

（2）理解开发部属。管理者既要了解部属的技术能力，也要了解他们的个性特点。对技术能力的了解可以帮助管理者分配工作，而对个性特点的了解则可以帮助他指派工作关系。

（3）提供清晰的短期目标。应该为项目指定容易达到的目标。长期目标是必要的，但并不能

很成功地激励项目人员每天的工作。管理者必须保证设置了短期的、高度确定的目标而且部属也清楚应如何达到这些目标。

（4）拥有共同焦点。项目人员对项目必须具有共同的目的，才能保证他们都在为共同的目标工作。管理者要建立和培养这个焦点。

（5）寻找对过程的改进。认识到每个项目都和以前的项目有微小区别之后，管理者必须监控开发过程，并在必要的时候和地方加以改进。虽然这些往往只是让特定过程更有实效的微小调整，但它们可以显著提高生产率。 247

（6）推动交流。当出现“热点”问题导致争论时，决定前进路线的最佳方式是召开受推动的会议。可以这样做：

- 确定关键参与者。
- 收集争论中的证据。
- 对存在哪些问题达成清晰的一致看法。
- 确保听取了所有人的意见。
- 确认所有人都理解了解决方案的可选项。
- 就首选方法达成一致。
- 尽可能让对此感到顾虑的人员参与实现确定的解决方案。

（7）管理交流机制。一般而言，电子邮件和新闻组是有用的交流机制，但是两者都可能让人处于随时可能毁灭的崎岖道路上。电子邮件可能迅速失控，而新闻组可以产生自大的、攻击性的争论。解决方法是每天跟踪电子邮件和新闻组上贴出的消息，确定所有的错误交流。直接与涉及的人进行交谈，让他们停止电子争论。如果事情很重要必须找到解决方案，那么受推动的会议通常是更快而且更有效的方法。

（8）异常管理。走极端总是危险的，过管理也不例外。要求管理者参与项目中所有线程的每日例会和每周回顾就是过管理或微管理的例子。应该根据异常进行管理。观察，但是不干预。给问题留出一些时间，让其他人可以解决它。只在必要的时候才插手。

（9）采用有效的决策制定方法。Kepner–Tregoe提出的两种理性管理方法在软件决策制定中尤其有效。第一种方法被称为情况分析[Kepner 1981]，该方法可以帮助组织和管理无组织的混乱环境。第二种方法被称为决策分析[Kepner 1981]，该方法对客观地制定决策很关键。决策制定过程中的主观偏见会对软件项目产生灾难性后果。我们经常会见到这类偏见导致软件灾难。友谊、销售电话打入的时机、不必要的平台限制和不可见的基础结构限制都可能会产生偏见。 248

接下来我们将更深入的说明这两种方法。

**情况分析**

情况分析的目的是在复杂情况中确定具有最高优先级的关注点。该方法经过裁减的版本包括下列步骤：

（1）列出所有关注点。这些关注点可能包括已知问题、行动事项、承诺、技术缺口和其他的事项。也就是说，列表中实际上可能包含所有可以影响到当前项目的事项。

（2）根据三个准则：重要性、迫切性和成长潜力来评定所有关注点的等级。使用高等、中等和低等这三个等级。例如，如果该关注点对项目成功是关键的，那么它就具有高等级的重要性。如果必须立即解决该关注点才能让决定有效，那么它就具有高等级的迫切性。如果预期重要性会随着时间显著增加，那么就认为它具有高等级的成长潜力。

（3）根据等级列出所有事项的计分表。为不同等级分配计分值：高等＝2、中等＝1，低等＝0。例如，如果三方面的等级分别为高等/中等/低等，最后的得分就是3分（2＋1＋0）。

（4）根据得分确定事项的优先级。对所有得分为6的关注点，依次选择最重要的、次重要的，如此等等。继续对得到5分或更低分数的事项分配优先级，直到列表中的所有关注点都被分配了优先级编号。

（5）把可以解决的事项分配给适当的部属。列表中的某些事项是由外部力量控制的，除非情况发生变化，否则花精力解决它们显然是不合理的。由于依赖关系，剩下的某些事项可能必须先于其他事项被解决。对于具备适当资格的部属，某些问题的解决将是直截了当的：编写必要的对象、建立分析模型、通过测试减少风险，等等。管理者应该让人力资源和必须解决的关注点达到
249 最佳的匹配状态。

（6）处理列表中具有最高优先级的事项。情况分析方法的工作原则是首先解决最重要的事，其次的事情永远不处理。通过使用情况分析，管理者可以从软件项目所面临的令人混乱的关注点列表中找出最重要的事项有效地加以解决。

**决策分析**

裁减过的决策分析过程包括下列步骤：

（1）定义要处理的决策的范围。也就是说，该过程开始于一个书面写好的待解决问题。

（2）确定解决该决策的可选解决方案。可以包括详细的具体可选方案，例如可以评估的特定软件产品或解决方案种类。

（3）定义决策准则。可以使用情况分析来建立此列表。

（4）把准则划分成必备准则和合意准则。必备准则就是方案要被接受就必须具备的那些特性。如果某个可选方案中缺少任何必备准则，它就失去了资格。因此，必备准则应该是仔细选择过的简短列表。所有其他准则都被称为合意准则，要确定它们的优先级。使用情况分析中的优先级分配过程来完成这个工作，并给它们分配权重。例如，如果有7个准则，则最重要的准则权重是7，次重要的准则权重是6，等等。

（5）确定可选方案是否满足所有必备准则。如果不满足，就去掉不能满足准则的方案。

（6）进行事实发现。对可选方案进行研究，分别评估它们对准则的满足程度。

（7）以准则和可选方案分别为纵横轴建立表格。在表格中，根据合意准则给可选方案排序。例如，如果有5个可选方案，最佳的方案得到的序号是5，其次的是4，等等。

（8）用权重乘以序号得到矩阵所有位置的得分。把每一列的得分加到一起计算出可选方案的

总分。总分最高的可选方案就是理性的（或者说最好的）选择。

（9）要注意理性选择不一定是管理者或客户会接受的选择。重要的偏见或在决策分析中未能 250
正确反映的准则都可能会导致这种情况，这时重要的是对决策准则进行评估。对合意准则的权重进行不同的试验并且增加新的准则，直到决策分析选择出了可接受的方案。确认最终的准则和权重符合驱动该解决方案的主观决策准则。在决策制定者们面对他们的偏见对决策的实际影响时，也许会把他们的准则改变成更现实的假设。

### 7.6.7　变化

顾问是无价的资源。他们可以为机构带来缺少的知识和技能，并提出独立于内部政策和Detailitis的建议。也就是说，他们可以在紧张的情况中保持客观。顾问在软件项目中可以扮演三个关键角色。这是根据Block关于一般顾问实践的模型得到的分类[Block 1981]：

（1）一双手。
（2）技术专家。
（3）管理同伴。

担任一双手的角色时，顾问可以成为与其他普通雇员相似的软件开发人员。作为技术专家时，顾问解决特定领域中的问题。作为管理同伴时，顾问是管理层的指导者，根据他的相关经验提供指导意见。虽然让顾问处于后两个角色中是有专业价值的，我们也曾看到他们在“一双手”的角色中提供了突出的结果，尤其是在使用新技术的快速原型构造中。

### 7.6.8　示例

我们曾经共事过的一名管理人员在一个项目的中途才加入。他并不了解部属的技能，也不清楚项目当前的背景。发生问题时，他对部属进行了重新分配。这产生了更严重的问题。部属成员调动到他们从未接触过的项目，导致了效率的降低。该管理者离开公司以后，情况得到了部分解 251
决。新来的管理者是非常善于处理人员关系的人，他保证让雇员的技能和必要的活动相匹配。然而，已经造成的延误是无法弥补的。

在另一个情况中，项目架构师（他是个Corncob）通过电子邮件向项目的所有成员表达了他对C++编码标准的看法。项目中有几个大师级的C++开发人员反对该架构师的某些说法。这导致在电子邮件和新闻组中进行了两个月时间的论战。最终解决方案是项目经理指定了一支“飞虎队”来解决这个问题。这是基于run-ahead原则的受推动的解决方案。不过，没有采取预防措施来在许多其他主题上消除类似的破坏性交流。处理措施只是治标而没有治本。

在第三个例子中，系统的集成项目有一个非理性管理团队。每个管理者都是他们那部分过程的所有者，这造成了相互之间既不一致也没有联系的烟囱过程，如图7-8所示。这导致了多次争夺所有权的战争。其中一次战争是关于从开发的一个阶段移动到另一个阶段的进入准则。它造成

了一个月时间的项目拷问和争论，分散了对开发中出现的问题的注意力。公司管理层不得不介入。由于解决方案是由上层强制实施的，它并没有支撑很久。

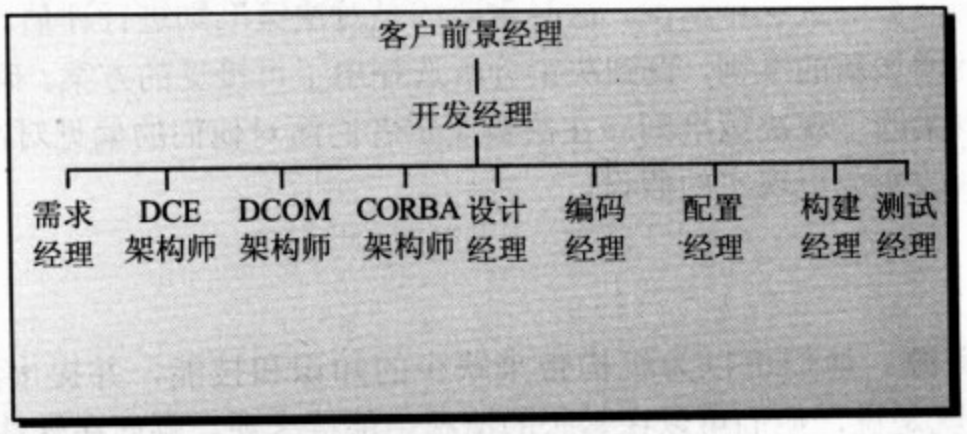



图7-8　非理性管理团队

在所有者之间的数次战争之后，最终解决方案是指定一个过程推动者来保证：

- 形成一致的、连续的过程。
- 对开发的每个阶段的进入准则达成一致，从而支持迭代式交付。
- 跨越相关过程实现对过程的改进。

252

## 小型反模式：Smoke and Mirrors（虚幻形象）

### 反模式问题

演示系统是重要的促销工具，因为它们常常被最终用户看作代表了具有产品质量的能力。急于开拓新业务的管理团队有时会（无意地）鼓励这些误解，做出的承诺超过了机构所能够交付的可运行技术的能力。这让开发人员处于非常艰苦的环境，因为他们受到压迫要交付许诺的能力。最后的输家是最终用户，他们无法在许诺的时间和成本下收到期望的能力。或者在系统交付时，最终用户常常会收到一个匆忙赶制的Stovepipe System，这个系统与演示环境相似，但是缺乏可维护性。

该反模式的首要原因被称为雾件（vaporware），它与期望管理有关。最终用户受到引导相信可以在实际上不能交付的时候交付某些能力，或者至少不是在预定的功能、预算和进度要求内交付。当开发项目无法交付时，项目会被看作是不成功的，而且开发机构的信誉会遭受损失。

### 重构方案

对最终用户期望的管理在道德和维护信誉方面都是重要的。软件工程的一条典型规则就是可交付系统花费的时间是工程原型花费时间的3倍。一组可复用软件花费的时间则是可交付系统所需软件的3倍。调查证明的另一条规则是任何软件开发花费的时间和成本都大约是预期的2倍[Johnson 1995]。

期望管理往往意味着最好让大家期待的内容少于可以交付的。当超出期望时，接收者会得到意外的惊喜，更可能成为常客。

**变化**

当Smoke and Mirrors是商业产品的演示时，可以采取一些有效的附加预防措施。有一种策略是让具备适当能力的工程师参加供应商的培训课程。通常，培训课不属于市场交流程序，因此会传递有关该产品的未经过滤的信息。但适当的工程师可以确定产品的真实能力并回报给开发团队。另一个成本较低的方案是要求对方提供一份产品技术文档，然后由合适的工程师进行评审。还要仔细评审安装需求和软件许可协议。安装需求可能会要求与目标环境不兼容的操作系统版本和产品版本。几乎没有软件许可会对适销性提供保证。也就是说，产品可能对它能够执行有用的功能没有提供法律上的保证。可以在购买协议中增加这样的保证以及扩展的产品支持。这样的保证在通信等行业中是很平常的事，在这些行业中可能会提供长达80年（系统的整个生命周期）的产品支持。

253

254

反模式

## 7.7　Project Mismanagement（项目管理不善）

**反模式名称**：Project Mismanagement
**别名**：Humpty Dumpty（矮胖子）
**最常见规模**：企业层
**重构方案名称**：Risk Management（风险管理）
**重构方案类型**：过程和角色
**根源**：责任（通用原因）
**不平衡的力量**：风险管理（通用作用力）
**轶事证据**："出了什么问题？所有事情都不错然后突然的……轰隆一声！"

### 7.7.1　背景

该反模式涉及对软件项目的监督和控制。它在时间上起始于规划活动之后，覆盖软件系统的实际分析、设计、构造和测试过程。Project Mismanagement主要是假定没有犯下任何规划错误（例如Death by Planning）的情况下在项目日常运行中犯下的错误。

### 7.7.2　一般形式

项目的运行是和建立项目计划同样复杂的活动，而且开发软件就像建造摩天楼一样复杂，同样涉及许多步骤和过程，包括进行检查和权衡。而现实中，关键活动常常被忽视或减少到了最少的量。这些关键活动包括技术规划（架构）和质量控制活动（评审和测试）。特别是基本的错误主要包括：架构定义不够充分、代码审查（软件评审）不足以及测试覆盖不充分。在测试这个总
255 体下有数个必要的阶段，但是常常被最小化了。它们分别是单元测试、集成测试和系统测试。简单的集成测试主要是对经过单元测试的构件的互用性进行基本检查。完整的集成测试会检查所有已知的成功路径和错误状态。

"你在生命中需要的就是无知和信心，然后就必然取得成功。"

——马克·吐温（美国著名作家）

架构为项目建立详细的技术计划，包括评审和测试的准则。当定义的是不充分的架构时，就没有为在评审和测试阶段对设计进行检查提供足够的基础。进行测试时，可能按照架构集成了软件模块，但无法根据应用的需要进行互用。

### 7.7.3　症状和后果

- 由于缺乏架构策略，设计难以实现。
- 不经常进行代码审查和评审，只带来了很少的价值。
- 由于没有充分定义系统的行为指南，测试设计需要进行额外的工作和猜测。
- 在集成测试和系统测试阶段，由于大量本应在之前的阶段如架构、设计、评审和单元测试中去除的缺陷，对项目进行大量的“拷问”。
- 开发将要结束，快要进入交付/接受阶段时，缺陷报告不断增加。

### 7.7.4　典型原因

- 不充分的架构未能定义用于评审、测试、集成和互用的技术准则。
- 代码审查和软件评审未评估设计缺陷，因此必须在更为昂贵的阶段如接受性测试中对这些缺陷加以解决。
- 使用不充分的测试套件来检查基本集成要求，但没有解决应用的全部互用要求。 256
- 所有前述因素都标志着低效的风险管理，可以上溯到架构师、设计者和管理层的专业实践中。

### 7.7.5　已知例外

Project Mismanagement反模式从来都没有任何例外。

### 7.7.6　重构方案

正确的风险管理是解决Project Mismanagement反模式可预测的症状和后果的有效方法。可以按照几种有用的方式来对风险进行分类：管理风险、常见项目失败点[Moynihan 1989]和质量风险。为了更好地限制风险，就要理解这些分类。

**管理风险**

这些风险主要是由公司管理层造成和解决的：

- 过程。对产品开发的端到端定义。
- 角色。用于实现过程的特定责任。

**常见项目失败点**

这些项目关键风险是根据23个风险驱动器得到的：

- 成本超支。
- 项目提前终止。
- 开发错误的产品。

- ❑ 技术失败。

**质量风险**

- ❑ 程序和项目管理。规划和控制过程的有效性。
- ❑ 产品识别。问题定义的准确度。
- ❑ 架构定义。设计规范和编码策略。
- ❑ 方案设计。交付完整支持解决方案的一致的、优化的编码说明的能力。
- ❑ 方案实施（编码）。产生设计的准确代码实现，让它以预期方式起作用的能力。
- ❑ 方案验证（测试）。证明实现的方案完全满足了问题的需求。
- ❑ 产品支持。维护和增强已发布产品的持续能力。

257

**共同理解**

风险管理中的主要问题常常是缺乏共同理解，导致对开发的看法不一致，引起解决方案不能满足问题需求范围的风险。公司必须了解它在所有场所和所有项目上进行的软件开发。每个项目都必须分享有关它的开发的一般知识。

理想情况下，一个项目中的所有（或大部分）开发人员对问题需求和意图实现的解决方案都应有合理的整体理解。不过，事实往往并非如此，从而导致下面的某些失败：

- ❑ 缺少功能。由于对要交付的平台上所有构件间将如何交互没有共享的视图，某些构件服务从来没有被建立起来。
- ❑ 错误功能。某些构件没有提供所要求的全部功能，因为开发人员从来没有与用户（其他开发人员）分享它的用途。
- ❑ 过功能。从来没有清楚地理解需求，从而从未摆脱“细节魔鬼”的困扰。这导致仅凭着“好意”进行软件开发，而没有清晰地建立任何需求或要求。
- ❑ 代码模块之间有接口但不能完全互用。没有定义跨越系统和模块边界的控制线程。

重构方案要求在开发的架构阶段和设计阶段进行一些活动。在架构层次，要定义模块之间的依赖。它包括分配定义模块间相互依赖的责任和分割系统的功能。在设计层次，要定义跨越模块的控制线程（用例）。这包括所有成功用例以及明确的错误状态。

258

## 7.7.7　变化

正规的风险管理活动是记录和规划风险缓解计划的一种方法。正规风险管理要求确定和记录项目风险。它通常采用表格的形式，包括由风险严重性、风险说明和风险缓解方法构成的列。风险管理计划往往是和项目的正式文档一起产生的，但很少会被转换成受资金支持的项目活动。重要的是确定项目风险的优先级并提供资金支持，对代表了关键失败点的风险缓解活动进行研究。

另一种变化形式则常常会使用一个前置团队来研究新产品的能力和技术的兼容性。前置团队

会使用首要技术或后备技术来建立类似于系统目标结构的原型。前置团队还会确定所有的技术缺点，研究绕过这些缺点的策略。然后该团队可以教育项目的其他成员有关该新技术的情况，从而缩短学习曲线。最后，前置团队的活动要领先于主开发活动1～3个月的时间。否则，主项目会由于要等待前置研究的结果而延误。

### 7.7.8　示例

某个软件开发没有对它的代码交付进行管理。开发团队被扔在一边，自行决定应如何把设计变成交付品。这意味着他们每个人都需要通过自生成的机制来处理质量和过程问题、管理他们自己的活动。由于缺乏对下列活动的坚持，风险变成了不可接受的代码：

- 代码标准（包括到设计的可追溯性）。
- 代码审阅。
- 测试规划。
- 单元测试。
- API测试。
- 集成测试。
- 功能测试。
- 程序文档。



风险以缺乏良好文档的代码的形式显示出来，而且这些代码大部分未经测试，没有集成，也不能完全满足需求。

指定新项目经理带来了质量的提高和过程的改进。改进是增量形式的。团队采取了一系列的“回顾”环节，在这些环节中确定问题然后建立特别小组，以增量过程改进的方法解决了这些问题。

每个特别小组都包括来自产品支持部门的推动者和下列过程的开发团队所有者（每开发团队一个）：

- 设计。
- 编码。
- 测试。
- 文档化。

特别小组的每个成员都要负责：

- 精炼特定过程。
- 教育他们的团队。
- 对团队实现该特定过程的情况和团队交付品的质量进行主观评价。

- 与小组其他成员保持联系来维护过程实现和解决问题的一致标准。

在6个月的时间里，每个过程都被增量地精炼了多次，每个团队也把每个过程实现了至少一次。改进效果是在下列方面的显著减少（超过50%）：

- 代码缺陷。
- 文档缺陷。
- 未测试的代码。

### 7.7.9　相关解决方案

260 Smoke and Mirrors反模式的一个变化说明了如何将后备技术结合到项目计划和活动中。这是对风险管理的重要补充，可以缓解Project Mismanagement产生的关键策略性错误。

Brad Appleton在他关于过程改进的论文中为改进软件开发过程介绍了一种优秀的设计模式语言。实现对过程的改进是解决Project Mismanagement的关键要素。

#### 小型反模式：Throw It over the Wall（各管一摊）

**轶事证据**

代码完成了（没有测试，没有文档）。

**背景**

文档很少会是可以完全自解释的，而理解作者的愿景和深入考虑是对文档进行的重要部分。指导性文档尤其如此，因为隐性地假设了它要独立地制定决策。该假设还意味着对作者意图的深入了解。

基于这种本质，所有人的知识都是个人的。即使最杰出的科学家也具有个人的见解来驱动他对新信息的发现和表达。理解这种个人见解对理解他们的工作非常重要。

**反模式问题**

太多的时候，下游管理者和面向对象开发人员会教条地对待本应作为灵活指导原则的面向对象方法、设计模式和实现计划。随着这些指导原则通过认可和公布，它们往往会被认为具有完备性和说明性（但实际上并未达到），并被强制地实现。

这种对灵活指导原则的教条解释会导致意图之外的结果。可能会基于本意只是为了激发认真分析和局部最优决策制定的指导原则来做出决策。例如，可能虽然开发团队中没人理解某些

261 分析和文档工作的目的，但由于它们看起来是强制性的，就会浪费许多精力在这上面。这种现象在大型机构和小型机构中都会发生，可能是由于开发阶段之间的错误信息交流而造成的。导致这一现象的另一个重要原因是为了满足管理层的明显愿望而不是满足系统最终用户的需要。

**重构方案**

要让技术文档被按照它的意图来理解和实现，必须使用一些重要的方法来交流这些材料。方法之一是通过教学课程传递这些知识。只要建立了新的政策和指导原则，就应该对这些信息

进行相应的传递来让大家了解这些信息及其动机。

我们发现1天的培训对多达100页的开发说明已经足够了。不过，按照两个阶段来进行培训可能是有益的：一个管理层介绍和一个针对开发人员的技术简报，因为这两群不同听众的需要存在显著的差异。如果把这两群人混到一起，管理方面的讨论常常会延伸占用技术细节所需的时间。包括电话和电子邮件联系在内的后续支持对保证成功的技术转移是有益的。

## 小型反模式：Fire Drill（消防演习）

### 反模式问题

Fire Drill是在很多软件开发机构中反复出现的场景。一个项目启动了，但是项目成员把设计和开发活动延迟了几个月来在管理层次上解决许多非技术问题(有一个软件开发人员把工作中的软件交付策略描述成："一直等到管理层感到绝望的时候，这时他们就会接受你给他们的任何东西。")。管理层或者告诉开发人员进行等待，或者给出一些不确定的或相互矛盾的指示，让他们不能前进。也许最具有破坏性的是外部产生的对项目方向的改变，这会导致返工和无法取得进展。

在进入项目进度表几个月之后，管理层才会意识到必须让开发马上取得进展。项目将会被取消是最常见的激发因素。在开发人员全体参加的Fire Drill午餐会上会宣布这一情况，这时管理层会对软件交付提出野心勃勃（或者说不切实际）的要求。典型的例子是项目花了6个月的时间进行需求分析和规划，然后在不到4个星期的时间内完成设计、实现和软件演示。

由于整个项目时间紧迫，自然就要在软件质量和测试方面做出妥协。环境的急迫性以一种不正当的方式让某些软件开发人员的工作变得更简单，因为在落后于进度的情况下，管理层会接受几乎任何软件（或文档）产品而不会提出什么问题。然而，在最终期限之前交付产品的那些尽责的开发人员常常会被管理层强迫对他们的解决方案进行返工。

### 重构方案

项目管理层可以实施的一个有效方案被称为sheltering。无论有哪些未解决的管理层问题，项目管理者都要负责交付软件产品。工作的环境要求与Fire Drill环境存在显著差异的高质量软件开发。架构驱动的开发尤其要求长时间期限和长期承诺。根据Booch（1996）和其他权威的看法，架构驱动的开发是取得软件成功的最有效方法。与之相反，Fire Drill环境让项目人员失去持续力，而这对当今的软件管理人员而言是很重要的。

在sheltering解决方案中，管理层建立和维护两个可选项目环境：内部的和外部的。软件开发人员的主体在内部环境中工作，其关注点是长期的，鼓励朝向软件交付的持续进展。系统中多达80%的软件不是应用特定的（所谓的内部模型）[Mowbray 1997c]。内部项目环境可以逐步发展到构造独立于外部技术非技术情况变化影响的内部模型。构建内部增量也是在迭代增量式项目中对软件开发资源的最有效利用[Korson 1997]。

外部环境也被称为项目的"公众形象"。它的作用是维护与外部实体，如高层管理、客户和姊妹项目（既存在资源竞争也存在复用机会）等的关系。外部环境的人员可能需要反复制造项目危机来获得重要资源。在Fire Drill文化中，当项目之间就管理层的重视和资源展开竞争时，

紧急情况会被看作等同于确定获得认可。少量管理人员和开发人员就可以维护外部环境。他们的工作是应对外部环境中的变化，以便项目人员的主体可以受到保护。外部环境中的活动的例子包括进展报告、状态回顾、采购、人事管理、客户展示和推销演示。Sheltering可以有效地把大部分项目成员和这些活动隔离开。

不过，有些时候紧急事件是真实的，可能会需要对开发时间和投入的努力做出英勇的承诺。不过，限制Fire Drill的频度很重要，这样开发人员才能处理真实紧急事件的要求。

**相关解决方案**

Fire Drill小型反模式与几个其他的关键反模式有关，这些反模式包括Analysis Paralysis、Viewgraph Engineering和Mushroom Management。在Analysis Paralysis中，在分析建模过程中要求完美导致分析阶段的拖延，压缩了开发进度表，从而产生Fire Drill。在Viewgraph Engineering中，机构从来没有从纸面分析转换到软件开发，工作环境与Fire Drill午餐会前的环境非常相似。在Mushroom Management中，开发人员与实际的最终用户被不必要的隔离开（不是最终用户管理）。开发人员无法获得清晰的需求或对用户界面能力的反馈。Mushroom Management的解决方案要求在最终用户和开发人员间进行建设性的交流，而Fire Drill的解决方案要求在方向的变化和不确定性可能让软件开发过程脱轨时，在开发人员和最终用户管理间进行破坏性的交流。

264

## 小型反模式：The Feud（管理不和）

**反模式问题**

该反模式也被称为Dueling Corncobs、Territorial Managers和Turf Wars。The Feud的标志是管理者之间的个性冲突，这会对工作环境产生严重影响。由于管理者之间的敌视会在他们的部属的态度和行为中反映出来，这些管理者的部属常常要承担他们的主管之间的争执造成的后果，而这些后果总是负面的。

因此，软件开发人员必须忍受缺乏建设性的交流，而总体上缺乏合作阻碍了任何形式的有用的技术转移。其后，公司的生产率和形象会遭受负面影响。

当对抗爆发成激烈的语言冲突时，会开始在高层管理人员面前对其他职员进行造谣中伤。这样的行为会涉及到整个公司管理机构，浪费时间和精力。电子邮件的对抗会导致冲突的严重恶化（请参阅E-mail Is Dangerous小型反模式）。管理层的不和会拖延数年，如果没有迅速解决的话，就会导致公开的敌对状态在很长时间内反复出现。

**重构方案**

Randall Oakes博士是富有很多信息技术迁移经验的人，他曾说过：“没有匹萨聚会不能解决的问题”。 [Oakes 1995]。他的意思是常常可以在友好的办公室聚会中解决组织问题。

我们在一些情况下使用过匹萨聚会的方法，并获得了积极结果。这些活动可以起到如同破冰船的效果，可以鼓励团队建设、推动友谊的形成和建立跨越组织边界的交流。

匹萨聚会可能对提供经济赞助的人最有好处。在匹萨聚会后，同事们会以新的眼光来看这个人：“他毕竟不是那么个怪物。他给我们买了匹萨，我们过得不错。”匹萨聚会的赞助者就可

以改变他们的形象。

**变化**

公司干预是解决机构性差异的一种方法。专业会议推动者和许多心理学家都实践过这些方法[GDSS 1994]。通过使用电子会议工具，两天的不在场干预可以获得显著的效果。这些会议可以帮助机构通过使用不在场团组的创造性来解决他们的差异，从而“彻底改造公司”。会议可以推动固执的管理者进行交流，建立新的关系。与会者会为以前看起来无法解决的问题提出创新性的解决方案。

265

## 小型反模式：E-mail Is Dangerous（电子邮件危机）

**别名**

Blame-Storming（谴责风暴）。

**反模式问题**

电子邮件是软件开发人员的重要交流媒介。不过，对很多主题和敏感的交流来说，它并不合适。例如，电子邮件不适合大多数对抗性的讨论。在电子邮件争论中，很容易发怒，伤害感情。更糟的是，电子邮件会让争论变成公众事件。如果软件项目成员被长期的电子邮件对抗缠住，项目的产出率和士气会迅速降低。

这个小型反模式也被称为E-mail Flaming，它会导致多种负面后果。下面列出了一些这样的后果，以及建议的预防措施：

- 一条“秘密”消息很可能最后会进入你最不希望读到此消息的人的收件箱。最好是把每封电子邮件都看作会直接落到你最敌对的对手和最强硬的竞争者手中。

266

- 电子邮件可以同时发给许多人，例如整个部门、公司、客户邮件列表和公开的因特网论坛。把每封电子邮件都看作它会被发表在华盛顿邮报上。
- 电子邮件消息可以成为永久的书面记录。把每封电子邮件都看成它会被当作法庭上的证据。

电子邮件是交流复杂主题的低效方式。由于电子邮件媒介的技术特点和其他关键特性，它容易受到误解，因为电子邮件的大量交换会把讨论降低到最小公分母的水平。而且，电子邮件讨论组会在各种主题上贴出数以十计的帖子，包括琐细的和不重要的主题在内。这些冗长的讨论既耗费时间也耗费大量人力。

**重构方案**

按照建议的那样谨慎地使用电子邮件。避免把电子邮件用于下列类型的消息：争论、批评、敏感信息、非技术上不合适的主题和可受到法律指控的声明。如果对使用电子邮件是否合适抱有任何怀疑，就使用其他的媒介。虽然电话谈话、传真件和面对面的讨论的内容同样可能会被揭露，但它们造成破坏的潜力远没有那么严重。

267

# *Part 3*

# 第三部分

# 结论和资源

本部分内容

- 附录 A　反模式大纲
- 附录 B　反模式术语表
- 附录 C　缩略语
- 附录 D　参考文献

# 附录 A 反模式大纲

表A-1中对本书中的反模式进行了归纳，小型反模式归纳于表A-2中。最右侧的所在章一栏指出了该反模式是在哪一章说明的。

表A-1 反模式摘要

| 反模式名称 | 反模式方案 | 解决方案 | 所在章 |
| --- | --- | --- | --- |
| Analysis Paralysis（分析瘫痪） | 在分析阶段力争达到完备和完美，导致项目停滞不前 | 增量、迭代式开发过程把进行详细分析的时间推迟到获得需要的知识之后 | 7 |
| Architecture by Implication（实现主导架构） | 系统开发没有文档化的架构，往往是由于最近的成功而过于自信 | 按照对应于系统利益相关者的多个视角来定义架构 | 6 |
| The Blob（胖球） | 过程式风格的设计导致一个对象具有大量责任，而大部分其他对象只保存数据 | 重构设计让责任分布得更均衡，并隔离变化的影响 | 5 |
| Corncob（玉米棒子） | 难于相处的人阻碍和偏转了软件开发过程 | 通过各种战术的、战役的和战略的组织行动来解决这些人的日程表 | 7 |
| Cut-and-Paste Programming（剪贴编程） | 通过复制源语句进行的代码复用导致显著的维护问题 | 通过在多次复用中使用共同的源代码、测试和文档来建立黑盒复用，减少维护问题 | 5 |
| Death by Planning（规划致死） | 对软件项目的过度预先规划导致开发工作的推迟和无用的计划 | 采用迭代式软件开发过程，包括使用已知事实进行适度的规划和增量式的重规划 | 7 |
| Design by Committee（委员会设计） | 委员会设计过于复杂，而且缺乏共同的架构前景 | 安排适当的推动措施和软件开发角色，以获得更有效的基于委员会的过程 | 6 |
| Functional Decomposition（功能分解） | 使用面向对象语言和标记方法来编码非面向对象的设计（可能来自于遗留设计） | 使用面向对象原则进行再设计。没有直接的重构方法 | 5 |

（续）

| 反模式名称 | 反模式方案 | 解决方案 | 所在章 |
|---|---|---|---|
| Golden Hammer（金锤） | 熟悉的技术或概念被强迫性地应用到很多的问题中 | 通过教育、培训和读书研讨会来扩展开发人员的知识，让他们了解新的解决方案 | 5 |
| Irrational Management（非理性管理） | 习惯性的优柔寡断和其他的习惯导致事实上的决策和开发中的紧急事件 | 利用理性决策制定管理方法 | 7 |
| Lava Flow（岩浆流） | 死代码和遗忘的设计信息冻结在不断变化的设计中 | 使用配置控制过程来消除死代码，为了提高质量而发展/重构设计 | 5 |
| Poltergeists（恶作剧鬼） | 类只有非常有限的职责和生命周期，常用于启动其他对象的处理过程 | 把职责分配给生命更长的对象，消除这些Poltergeists类 | 5 |
| Project Mismanagement（项目管理不善） | 对软件开发过程管理的疏忽导致漫无方向和其他症状 | 对软件开发项目进行监控以进行成功的开发活动 | 7 |
| Reinvent the Wheel（重新发明轮子） | 具有重叠功能的遗留系统不能互用。每个系统的构建都是相互隔离的 | 使用架构挖掘和“优化杂交”概括方法来定义通用接口，然后使用对象封装来集成 | 6 |
| Spaghetti Code（面条代码） | 即兴开发的软件结构导致难以扩展和优化代码 | 频繁地重构代码以改进软件结构、支持软件维护和迭代式开发 | 5 |
| Stovepipe Enterprise（烟囱企业） | 未经协调的软件架构导致缺乏可适性、复用和互用性 | 使用企业架构规划来协调系统开发协定、复用和互用 | 6 |
| Stovepipe System（烟囱系统） | 即兴开发的集成方案和缺乏抽象导致脆弱的、不可维护的架构 | 使用抽象、子系统外观和元数据来生成可适的系统 | 6 |
| Vendor Lock-In（供应商锁定） | 专有的、依赖于产品的架构未对复杂性进行管理，导致失控的架构和维护成本 | 在依赖于产品的接口和应用软件主体之间安装一个隔离层，让对复杂性和架构的管理成为可能 | 6 |



表A-2　小型反模式摘要

| 反模式名称 | 反模式方案 | 解决方案 | 所在章 |
|---|---|---|---|
| Ambiguous Viewpoint（横糊视角） | 不清晰的建模视角导致在对象模型中出现造成问题的模糊 | 澄清是对下面的哪一个基本视角进行建模：业务视角、规范视角或者实现视角 | 5 |
| Autogenerated Stovepipe（自生成烟囱） | 从细粒度头文件为分布式大规模系统自动生成接口 | 把架构层框架设计和子系统特定的设计隔离开以管理复杂性 | 6 |
| Blowhard Jamboree（吹牛者狂欢） | 行业中的权威人士散布行销信息引起客户的顾虑 | 安排内部专家来把事实和欺骗宣传区分开 | 7 |
| Boat Anchor（船锚） | 系统开发项目购买了一个高成本的技术，但是没有使用它 | 在购买之前委派合适的工程师对产品进行评估 | 5 |

（续）

| 反模式名称 | 反模式方案 | 解决方案 | 所在章 |
| --- | --- | --- | --- |
| Continuous Obsolescence（持续过时） | 因特网时代的技术发布速度超过了跟上技术进步与之保持同步的能力 | 依赖你能够控制的稳定技术和接口。开放式系统标准提供了稳定性 | 5 |
| Cover Your Assets（隐藏资产） | 文档驱动的软件开发过程中的文档作者常常只列出可选方法而不制定决策 | 为文档化任务建立清晰的目的和指导原则，评审其结果以了解文档化决策的价值 | 6 |
| Dead End（死胡同） | 对商业软件或可复用软件的直接修改让软件系统产生了显著的维护负担 | 避免修改受外部支持的软件。尽可能选择主流的、受支持的产品和平台 | 5 |
| E-mail Is Dangerous（电子邮件危机） | 电子邮件是有用的、但是不稳定的交流方式 | 避免使用电子邮件传递敏感的、争论性的或对抗性的信息 | 7 |
| Fear of Success（成功恐惧症） | 在项目接近成功完成时，大家（包括软件开发人员）做出一些疯狂的举动 | 项目即将完成时，明确地宣布取得了成功 | 7 |
| The Feud（管理不和） | 长期与同僚冲突的管理者对他们的部属有严重的负面影响 | 使用专业的推动手段或非正式聚会来解决分歧 | 7 |
| Fire Drill（消防演习） | 管理层等到最后可能的时刻才让开发人员开始进行设计和实现，然后他们几乎立即就想获得结果 | 即使客户和管理层不能全体在场，也要主动开始进行设计和原型构造 | 7 |
| The Grand Old Duke of York（约克老公爵） | 4/5的开发人员不能定义良好的抽象。这导致过度的复杂性 | 指定抽象派担任项目团队架构师——他们是具有架构本能的人 | 6 |
| Input Kludge（输入拼凑） | 定制编写的输入算法含有很多错误，这些错误对用户和测试人员都显而易见 | 利用具有产品质量的输入处理方法，包括词法分析、解析生成器和功能矩阵 | 5 |
| Intellectual Violence（智力暴行） | 大家为了胁迫他人或获得短期利益而晦涩地引用深奥的文献、理论和标准 | 在整个机构中鼓励进行教育和实践指导 | 7 |
| Jumble（混乱） | 接口设计是对横向要素和纵向要素未加考虑的混合，使得必须频繁改变接口，而且无法复用 | 按照横向要素、纵向要素和元数据要素来划分架构设计 | 6 |
| Mushroom Management（蘑菇管理） | 开发人员处于很暗中被人“喂养”。禁止他们与最终用户进行交互 | 经常和用户进行交互以让可用性和用户接受性最大化 | 5 |
| Smoke and Mirrors（虚幻形象） | 最终用户错误地认为脆弱的演示系统展示的是可用于运行使用的能力 | 实施适当的规范来对处理销售和市场推广情况时的期望、风险、责任和后果进行管理 | 7 |
| Swiss Army Knife（瑞士军刀） | 对接口的过设计导致对象具有无数的方法，试图满足所有可能的需要。这导致难以理解、使用和调试的设计，同时还导致实现之间的依赖 | 为构件定义清晰的目的，适当地对接口进行抽象以管理复杂性 | 6 |

（续）

| 反模式名称 | 反模式方案 | 解决方案 | 所在章 |
|---|---|---|---|
| Throw It over the Wall<br>（各管一摊） | 生成和散发文档时并没有准备好技术转移。灵活的指导原则被错误地看作事实上的政策或正式的过程 | 保证把信息传递和散发到规划的任何新过程或指导原则的执行过程。包括指导性开发、培训的交接和技术转移工具 | 7 |
| Viewgraph Engineering<br>（视图设计） | 只看到了实际只能提供有限系统开发技术能力的机构的表面能力，因为他们可以产生大量文档和精美的简报 | 核实机构和关键项目人员的开发能力。利用原型和模型作为所有系统开发过程的组成部分 | 7 |
| Walking through a Mine Field<br>（穿越雷区） | 软件技术没有人们想像的那么稳健。错误非常普遍而且可能带来灾难 | 在软件测试和评审中进行投资，以减少软件缺陷的频度和密度 | 5 |
| Warm Bodies<br>（温暖身体） | 大型软件项目团队倾向于低效的组织结构和延期。英雄程序员很重要 | 规划小型项目（4个人用4个月时间）。他们更可能带来软件上的成功 | 6 |
| Wolf Ticket<br>（黄牛票） | 由于技术采用开放式系统包装或声称符合标准而被认为具有优良品质。几乎没有标准有测试套件（不到6%），而且很少会有产品真正测试过是否符合标准 | 找出隐藏在声明背后的真相；对权威提出质疑；不做出任何假设。把提供证明的负担转移给销售机构。直接与技术产品专家和开发人员进行交流 | 6 |

273

274

# 附录 B 反模式术语表

B

**Action Lever（行动杠杆）**：引起改变或解决问题的最有效机制。例如，进行性能优化时，通过测量发现的一小段导致性能瓶颈的代码就是一个行动杠杆。

**Also Known As（别名）**：反模式模板中的一个部分。包括被广泛使用的，或是能够描述该反模式的其他常见名称和短语。

**Anecdotal Evidence（轶事证据）**：反模式模板中的一个部分。用于描述此处出现的反模式的任何俗语或喜剧性材料。

**AntiPattern（反模式）**：确定会产生负面后果的经常发生的模式或解决方案。反模式可能是处于错误的上下文环境中的模式。正确文档化的反模式中包括一对方案，一个是反模式方案，另一个是重构方案。

**Applicability to Other Viewpoints and Scales（对其他视角和规模的适用性）**：反模式模板中的一个部分。它说明了该反模式如何影响其他的视角：管理视角、架构视角或开发人员视角。有时，该部分会包括该反模式对其他开发规模层次的令人感兴趣的指导。

**Architectural Benefits（架构效益）**：通过设计和使用良好的架构及相应软件接口带来的正面效果。典型效益包括可适性、成本和风险的降低，等等。

**Architectural Characteristics（架构特性）**：与设计制品相联系，影响到它的用途和在架构划分中的定位的特性。例如，成熟度、领域特异性、灵活性、限制、实现依赖、复杂性、稳定性，等等。

**Architectural Partition（架构划分）**：架构定义了不同设计制品类别之间的边界。这些划分把具有不同特性的实体类别分隔开。划分可以帮助描绘利害关系，减少相互冲突的作用力的数量，让问题更容易解决。划分还可以隔离可能独立改变的实体。例如，隔离开一个普通的可复用对象和领域特定的可复用对象。

**Architectural Placement Criteria（架构定位准则）**：设计模式是在问题陈述最适用的层次上进行说明的，而且解决方案的边界也处于该层次的范围内。这一定义有两个准则，其中问题的适用性优先于解决方案的范围。某些设计模式潜在地可以被安置于多个层次。模式模板的规模部分针对的就是模式在其他层次上的使用价值。

**Architecture（架构）**：对整个系统的多个视图。架构包括多个视图，每个视图分别来自系统中的一个潜在利益相关者，例如最终用户、开发人员、软件架构师、专家和管理者。

**Background（背景）**：反模式模板中的一个部分。这里出现的是与该反模式的所有其他部分的目的不一样的附加说明。

**Bytecode（字节码）**：介于Java等高级编程语言和机器码之间的中间表达方式。

**Component（构件）或Software Component（软件构件）**：处于应用层或更低规模层次上的小规模软件模块。在构件架构中，构件之间会共享共同接口和元数据以支持互用、构件替换和系统扩展。

**Design Artifact（设计制品）**：某个设计选择的特定实例。

**Design Pattern（设计模式）**：用于说明一个设计问题的预定义常识性解决方法的问题陈述和解决方案。正确文档化的模式使用一致的模板来进行说明，从而在保证简洁的同时保证对细节、相关事项和各种折中的全面覆盖。

**Design Point（设计点）**：设计模式中在允许的选项范围内做出的特定折中。特定问题的设计选项的完整范围形成了可用选择的闭集。设计点是这些选择之一，可以解决作用力，在效益和后果之间取得适当的平衡。例如，在IDL参数说明中选择字符串数据类型而不是枚举类型。枚举只有固定的可选项集，不对IDL做出改变就无法扩展，而字符串类型可以支持相当广范的使用和扩展。

**Example（示例）**：反模式模板中的一个部分。它给出该反模式及其重构方案的例子。

**Forces（作用力）**：与上下文环境相关的影响到设计选择的激励因子。在设计模式模板中的适用性部分确定了作用力，在模板的解决方案部分对其加以解决。参阅横向力量、纵向力量和原力。

**General Form（一般形式）**：反模式模板中的一个部分。它说明该反模式的一般特性。重构方案将解决这个部分中介绍的一般反模式。

**Horizontal Forces（横向力量）**：适用于多个领域或问题的作用力，会跨越数个软件模块或构件影响到设计选择。对于横向力量，其他地方做出的设计选择可能会对本地设计选择产生直接或间接的影响。

**Implementation（实现）**：组成提供符合某个接口的服务机制的代码（或软件）。也被称为对象实现（object implementation）。

**Interface（接口）**：在服务使用者（客户端）和服务提供者（实现）之间的软件边界。

**Java Virtual Machine（Java虚拟机）**：Java语言用来动态解释Java字节码的运行时系统。它还负责管理其他Java功能，例如垃圾收集和对象创建。

**Mining（挖掘）**：研究已经存在的解决方案和遗留系统，以便迅速获得稳健的理解来解决新问题。挖掘会引出对以前的解决方案进行复用的可能、对多个解决方案的横向概括，或者理解对遗留系统进行包装的需求。

**Module（模块）或Software Module（软件模块）**：对一部分软件的专业术语。模块被用于在不同规模上指代软件。应用层模块是子系统，而系统层模块是整个软件系统，等等。模块与同一规模上的其他模块是可以分离的。

**Most Frequent Scale（最常见规模）**：反模式模板中的一个部分。这里的规模是指软件开发层次模型（SDLM）中的一个层次，该反模式通常发生于这个层次的软件开发中。可选的规模包括：惯用法、微架构、框架、应用、系统、企业或全球/行业。规模限制了解决方案的适用范围。

**Name（名称）**：反模式模板中的一个部分。它是一个独特的名词或名词短语，故意采用贬义词。如果还有其他名称，就在别名部分进行说明。

**PLoP**：程序模式语言。有关建立模式和模式语言，以及对其进行文档化的年度会议。

**Primal Forces（原力）**：反模式模板中的一个部分。普遍存在于软件架构和开发中的那类横向力量。原力存在于大部分设计情况中，应该被看作驱动大部分解决方案的上下文环境作用力的一部分。

**Refactored Solution（重构方案）**：反模式模板中的一个部分。用于说明对该反模式的解决方案。这一部分与一般形式部分相互呼应。说明解决方案时不带有任何的变化，而且可以采用按照步骤说明的结构。

**Related Solutions（相关解决方案）**：反模式模板中的一个部分，包括用于解释该反模式和其他反模式之间差异的所有引用和交叉引用。

**Root Causes（根源）**：反模式模板中的一个部分，列出该反模式的一般原因。这些原因来自架构专栏文章“面向对象架构的死罪”[Mowbray 1997a]，采用了对圣经的隐喻。选项包括：匆忙、贪婪、自负、无知、思想狭隘和懒惰。被忽视的责任是通用的原因。

**Solution Type（解决方案类型）**：反模式模板中的一个部分。根据SDLM确定的反模式解决

方案所引发的行动的类型。可选的类型包括：软件、技术、过程或角色。“软件”指的是解决方案会产生新软件。“技术”指的是解决方案要求获取新的技术或产品。“过程”指的是解决方案需要采用某个过程。“角色”指的是解决方案要求将责任分配给个人或团组。

**Symptoms and Consequences（症状和后果）**：反模式模板中的一个部分。列出该反模式会导致的症状和后果。

**Template（模板）**：用于定义一个设计模式或反模式的解释性部分的大纲。模板的每个部分分别回答有关该模式或反模式的重要问题。

**Typical Causes（典型原因）**：反模式模板中的一个部分。和根源一起说明导致该反模式的独特原因。

**Unbalanced Forces（不平衡的力量）**：反模式模板中的一个部分。根据SDLM列出在该反模式中被忽视的、误用的或过分使用的一般作用力。可用的选项包括：功能管理、性能管理、复杂性管理、变化管理、IT资源管理和技术转移管理。风险管理是通用的作用力。

**Variations（变化）**：反模式模板中的一个部分。列出该反模式的所有已知主要变化。广为人知的替代解决方案同样在此加以说明。

**Vertical Forces（纵向力量）**：存在于某些特定领域或问题上下文环境中的情况特定的作用力。领域特定的力量在特定情况下是独特的。由于纵向力量对一个软件环境是独特的（或局部的），对纵向力量的解决通常会导致对每个软件问题采用独特的解决方案。完全根据纵向力量生成的接口被称为纵向接口（vertical interface）。

# 附录C 缩略语

| | | |
|---|---|---|
| ACID | Atomic, Consistent, Isolated, Durable | 原子的、一致的、隔离的、持久的 |
| ANSI | American National Standards Institute | 美国国家标准学会 |
| API | Application Program Interface | 应用程序编程接口 |
| CASE | Computer-Aided Software Engineering | 计算机辅助软件工程 |
| CD-ROM | Compact Disc Read-Only Memory | 只读光盘存储器 |
| CIO | Chief Information Officer | 信息总监 |
| CMU | Carnegie Mellon University | 卡耐基－梅隆大学 |
| COM | Microsoft Component Object Model | 微软组件对象模型 |
| CORBA | Common Object Request Broker Architecture | 分布式对象请求代理架构 |
| COSE | Common Open Software Environment | 通用开放式软件环境 |
| COTS | Commercial off-the-shelf | 商用现货 |
| DIN | German National Standards Organization | 德国国家标准组织 |
| ECMA | European Computer Manufacturers Association | 欧洲计算机制造商协会 |
| E-R | Entity-Relationship Modeling | 实体－关系建模 |
| FIPS | Federal Information Processing Standard | （美国）联邦信息处理标准 |
| FGDC | Federal Geographic Data Committee | （美国）联邦地理数据委员会 |
| FTP | File Transfer Protocol | 文件传输协议 |
| GOTS | Government off-the-shelf | 政府现货 |
| GPL | Gamma Pattern Language | Gamma模式语言 |
| HVM | Horizontal-Vertical-Metadata | 横向－纵向－元数据 |
| IBM | International Business Machines | 国际商用机器（公司名） |
| ICD | Interface Control Document | 接口控制文档 |
| IDL | ISO/CORBA Interface Definition Language | ISO/CORBA接口定义语言 |
| IEEE | Institute of Electrical and Electronics Engineers | 电子电气工程师协会 |

| | | |
|---|---|---|
| ISO | International Standards Organization | 国际标准化组织 |
| IT | Information Technology | 信息技术 |
| MVC | Model-View-Controller | 模型－视图－控制器 |
| O&M | Operations and Maintenance | 运行和维护 |
| ODMG | Object Database Management Group | 对象数据库管理组 |
| ODP | Open Distributed Processing | 开放分布式处理 |
| OLE | Microsoft Object Linking and Embedding | 微软对象链接和嵌入协议 |
| OLTP | Online Transaction Processing | 在线事务处理 |
| OMG | Object Management Group | 对象管理组 |
| ONC | Open Network Computing | 开放式网络计算 |
| OO | Object Oriented | 面向对象 |
| OOA | Object-Oriented Analysis | 面向对象分析 |
| OOA&D | Object-Oriented Analysis and Design | 面向对象分析和设计 |
| OOD | Object-Oriented Design | 面向对象设计 |
| OODBMS | Object-Oriented Database Management System | 面向对象数据库管理系统 |
| OQL | ODMG Object Query Language | ODMG对象查询语言 |
| OSE | Open Systems Environment | 开放系统环境 |
| OSF | Open Software Foundation | 开放软件基金 |
| OMA | Object Management Architecture | 对象管理架构 |
| PLoP | Pattern Languages of Programs Conference | 程序模式语言会议 |
| RFC | Request for Comment | 请求注释 |
| RFP | Request for Proposal | 请求提案 |
| SEI | Software Engineering Institute | （卡内基－梅隆大学的）软件工程学院 |
| SGML | Standard General Markup Language | 标准通用标记语言 |
| SPC | Software Productivity Consortium | 软件生产力联盟 |
| SQL | Structured Query Language | 结构化查询语言 |
| SYSMAN | X/Open Systems Management | X/开放系统管理 |
| TCP/IP | Transmission Control Protocol/Internet Protocol | 传输控制协议/网际协议 |
| TWIT | Third-World Information Systems Troubles | 第三世界信息系统问题 |
| URL | Universal Resource Locator | 通用资源定位器 |
| WAIS | Wide Area Information Search | 广域信息搜索 |

# 附录D 参考文献

正文中使用名字-日期标注方式引用了下列文献，例如[Katz 1993]。


[Adams 1996a] Adams, Scott, "The Dilbert Principle: A Cubicle's Eye View of Bosses, Meetings, Management Fads, and Other Workplace Afflictions," New York: HarperBusiness, 1996.

[Adams 1996b] Adams, Scott, "Dogbert's Top Secret Management Handbook," New York: HarperBusiness, 1996.

[Adams 1997] Adams, Scott, "Dilbert Future: Thriving on Stupidity in the 21st Century," New York: HarperBusiness, 1997.

[Akroyd 1996] Akroyd, Michael, "AntiPatterns Session Notes," Object World West, San Francisco, 1996.

[Alexander 1977] Alexander, Christopher, *A Pattern Language*, Oxford: Oxford University Press, 1977.

[Alexander 1979] Alexander, Christopher, *The Timeless Way of Building*, Oxford: Oxford University Press, 1979.

[Appleton 1997] Appleton, Brad, "Patterns for Conducting Process Improvement," PLoP, 1997.

[Augarde 1991] Augarde, Tony, *The Oxford Dictionary of Modern Quotations*, Oxford: Oxford University Press, 1991.

[Bates 1996] Bates, M.E., *The Online Deskbook*, New York: Pemberton Press, 1996.

[Beck 1996] Beck, Kent, "Guest Editor's Introduction to Special Issue on Design Patterns," *OBJECT Magazine*, SIGS Publications, January 1996, pp 23–63.

[Beizer 1997a] Bezier, Boris, "Introduction to Software Testing," International Conference on Computer Aided Testing, McLean, VA, 1997.

[Beizer 1997b] Beizer, Boris, "Foundations of Testing Computer Software," Workshop, 14th International Conference and Exposition on Testing Computer Software, Vienna, VA, July 1997.

[Blueprint 1997] Blueprint Technologies, "Software Silhouettes," McLean, Virginia, 1997.

[Block 1981] Block, Peter, *Flawless Consulting: A Guide to Getting Your Expertise Used*, San

Diego: Pfeiffer & Company, 1981.

[Booch 1996] Booch, Grady, *Object Solutions*, Reading, MA: Addison-Wesley-Longman, 1996.

[Bowen 1997] Bowen, Jonathan P., and Hinchey, Michael G., "The Use of Industrial-Strength Formal Methods," Proceedings of the Twenty-First Annual Computer Software and Applications Conference (COMPSAC 97), IEEE, August 1997.

[Brodie 1995] Brodie, Michael, and Stonebraker, Michael, *Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach*, Menlo Park, CA: Morgan Kaufmann Publishers, 1995.

[Brooks 1979] Brooks, Frederick P., *The Mythical Man-Month*, Reading, MA: Addison-Wesley, 1979.

[Brown 1995] Brown, Kyle, "Design by Committee," on the Portland Patterns Repository Web site, http://c2.com/ppr/index.html.

[Brown 1996] Brown, William J., "Leading a Successful Migration," *Object Magazine*, October 1996, pp. 38–43.

[Buschmann 1996] Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; Stal, Michael, *Pattern-Oriented Software Architecture: A System of Patterns*, New York: John Wiley & Sons, Inc., 1996.

[C4ISR 1996] C4I Integration Support Activity, "C4ISR Architecture Framework," version 1.0, Integrated Architectures Panel, U.S. Government Document CISA-0000-104-96, Washington, DC, June 1996.

[Cargill 1989] Cargill, Carl F., *Information Technology Standardization: Theory, Process, and Organizations*, Bedford, MA: Digital Press, 1989.

[Connell 1987] Connell, John, *Rapid Structured Prototyping*, Reading, MA: Addison-Wesley, 1987.

[Constantine 1995] Constantine, Larry, *Constantine on Peopleware*, Englewood Cliffs, NJ: Prentice - Hall, 1995.

[Cook 1994] Cook, Steve, and Daniels, John, *Designing Object Systems*, Englewood Cliffs, NJ: Prentice-Hall, 1994.

[Coplien 1994] Coplien, James O., "A Development Process Generative Pattern Language," PLoP, 1994.

[Coplien 1994] Coplien, James O., Object World briefing on design patterns, AT&T Bell Labs Conference Tutorial, San Francisco, 1994.

[Cusumano 1995] Cusumano, M.A., and Selby, R.W., *Microsoft Secrets*, New York: Free Press, 1995.

[Davis 1993] Davis, Alan M., *Objects, Functions, and States*, Englewood Cliffs, NJ: Prentice-Hall, 1993.

[Dikel 1997] Dikel, David; Hermansen, Christy; Kane, David; and Malveau, Raphael; "Organizational Patterns for Software Architecture," PLoP, 1997.

[Dolberg 1992] Dolberg, S.H., "Integrating Applications in the Real World," *Open Information Systems: Guide to UNIX and Other Open Systems*, Boston: Patricia Seybold Group, July 1992.

[Duell 1997] Duell, M., "Resign Patterns: Ailments of Unsuitable Project-Disoriented Software," *The Software Practioner*, vol. 7, No. 3, May–June 1997, p. 14.

[Edwards 1997] Edwards, Jeri, and Devoe, D., "10 Tips for Three-Tier Success," *D.O.C. Magazine*, July 1997, pp. 39–42.

[Foote 1997] Foote, Brian and Yoder, Joseph, "Big Ball of Mud," *Proceedings of Pattern Languages of Programming*, PLoP, 1997.

[Fowler 1997] Fowler, Martin, *Analysis Patterns: Reusable Object Models*, Reading, MA: Addison - Wesley 1997.

[Gamma 1994] Gamma, Erich; Helm, Richard; Johnson, Ralph; and Vlissides, John; *Design Patterns*, Reading, MA: Addison-Wesley, 1994.

[Gaskin 1979] Gaskin, Stephen, *Mind at Play*, Summerville, TN: The Book Publishing Company, 1979.

[GDSS 1994] Group Decision Support Systems, "Group Faciltation Using Groupsystems V," Training Course, Georgetown, Washington DC, 1994.

[Gilb 1993] Gilb, Tom, and Graham, Dorothy, *Software Inspection*, Workingham, UK: Addison - Wesley, 1993.

[Goldberg 1995] Goldberg, Adele, and Rubin, Kenny S., *Succeeding with Objects: Decision Frameworks for Project Management*, New York: Addison-Wesley, 1995.

[Griss 1997] Griss, Martin, "Software Reuse: Architecture, Process, and Organization for Business Success," Object World, San Francisco, 1997.

[Halliwell 1993] Halliwell, Chris, "Camp Development and the Art of Building a Market through Standards," *IEEE Micro*, vol. 13, no. 6, December 1993, pp. 10–18.

[Herrington 1991] Herrington, Dean, and Herrington, Selina, "Meeting Power," The Herrington Group, Inc., Houston, TX, 1991.

[Hilliard 1996] Hilliard, Richard; Emery, Dale; and Rice, Tom, "Experiences Applying a Practical Architectural Method." In *Reliable Software Technologies: Ada Europe '96*, A. Strohmeier(ed.), New York: Springer-Verlag, *Lecture Notes in Computer Science*, vol. 1088, 1996.

[Horowitz, 1993] Horowitz, Barry M., *Strategic Buying for the Future*, Washington DC: Libey Publishing, 1993.

[Hutt 1994] Hutt, Andrew (ed.), *Object Oriented Analysis and Design*, New York: John Wiley & Sons, Inc., 1994.

[ISO 1996] International Standards Organization, "Reference Model for Open Distributed

Processing," International Standard 10746-1, ITU Recommendation X.901, 1996.

[Jacobson 1992] Jacobson, Ivar, *Object-Oriented Software Engineering*, Reading, MA: Addison - Wesley, 1992.

[Jacobson 1997] Jacobson, Ivar; Griss, Martin; and Jonsson, Patrick; *Software Reuse: Architecture Process and Organization for Business Success*, Reading, MA: Addison-Wesley, 1997.

[Jacobson 1991] Jacobson, Ivar and Lindstrom, F., "Reengineering of Old Systems to an Object-Oriented Architecture," *OOPSLA Conference Proceedings*, 1991.

[Johnson 1995] Johnson, Johnny, "Creating Chaos," *American Programmer*, July 1995.

[Johnson 1993] Johnson, Ralph, "Tutorial on Object-Oriented Frameworks," *OOPSLA93 Tutorial Notes*, Association for Computing Machinery, 1993.

[Kane 1997] Kane, David; Opdyke, William; and Dykel, David; "Managing Change to Reusable Software," PLoP, 1997.

[Katz 1993] Katz, Melony; Cornwell, Donna; and Mowbray, Thomas J; "System Integration with Minimal Object Wrappers," *Proceedings of TOOLS '93*, August 1993.

[Kepner 1981] Kepner, C.H., and Tregoe, B.B., *The New Rational Manager*, Princeton, NJ: Kepner - Tregoe, Inc., 1981.

[Kitchenham 1996] Kitchenham, Barbara, *Software Metrics*, Cambridge, MA: Blackwell Publishers, 1996.

[Korson 1997] Korson, Timothy, "Process for the Development of Object-Oriented Systems," Tutorial Notes, Object World West Conference, July 1997.

[Kreindler 1995] Kreindler, R. Jordan, and Vlissides, John, *Object-Oriented Patterns and Frameworks*, IBM International Conference on Object Technology, San Francisco, CA, 1995.

[Kruchten 1995] Kruchten, Phillipe B., "The 4+1 View Model of Architecture," *IEEE Software*, November 1995, pp. 42–50.

[McCarthy 1995] McCarthy, J., "Dynamics of Software Development," Redmond, WA:Microsoft Press, 1995.

[McConnell 1996] McConnell, Steve, *Rapid Development*, Redmond, WA: Microsoft Press, 1996.

[Melton 1993] Melton J., and Simon, A.R., *Understanding the New SQL*, Menlo Park, CA: Morgan Kaufmann Publishers, 1993.

[Moore 1997] Moore, K.E., and Kirschenbaum, E.R., "Building Evolvable Systems: The ORBlite Project," *Hewlett-Packard Journal*, February 1997.

[Mowbray 1995] Mowbray, Thomas J., and Zahavi, Ron, *The Essential CORBA*, New York: John Wiley & Sons, Inc., 1995.

[Mowbray 1997a] Mowbray, Thomas J., "The Seven Deadly Sins of Object-Oriented Architecture," *OBJECT Magazine*, March 1997, pp. 22–24.

[Mowbray 1997b] Mowbray, Thomas J., "What Is Architecture?" *OBJECT Magazine*, Architecture

column, September 1997.

[Mowbray 1997c] Mowbray, Thomas J., and Malveau, Raphael C., *CORBA Design Patterns*, New York: John Wiley & Sons, Inc., 1997.

[Moynihan 1989] Moynihan, T.; McCluskey, G.; and Verbruggen, R.; "Riskman1: A Prototype Tool for Risk Analysis for Computer Software," Third International Conference on Computer - Aided Software Engineering, London, 1989.

[Oakes 1995] Oakes, R., Presentation at Healthcare Software Development Conference, Medical Records Institute, Boston, 1995.

[Opdyke 1992] Opdyke, W.F., "Refactoring Object-Oriented Frameworks," Ph.D. thesis, University of Illinois, Urbana, IL, 1992.

[PLoP 1994] *Proceedings of the First Conference on Pattern Languages of Programs*, August 1994.

[PLoP 1995] *Proceedings of the Second Conference on Pattern Languages of Programs*, August 1995.

[PLoP 1996] *Proceedings of the Third Conference on Pattern Languages of Programs*, August 1996.

[PLoP 1997] *Proceedings of the Fourth Conference on Pattern Languages of Programs*, September 1997.

[Pree 1995] Pree, Wolfgang, *Design Patterns for Object-Oriented Software Development*, Reading, MA: Addison-Wesley, 1995.

[RDA 1996] RDA Consultants, "Experiences Using CASE Tools on ROOP Projects," Tinomium, MD, 1996.

[Riel 1996] Riel, A.J., *Object-Oriented Design Heuristics*, Reading, MA: Addison-Wesley, 1996.

[Roetzheim 1991] Roetzheim, W.H., *Developing Software to Government Standards*, Englewood Cliffs, NJ: Prentice-Hall, 1991.

[Rogers 1997] Rogers, Gregory F., *Framework-Based Software Development in C++*, Short Hillzs, NJ: Prentice-Hall, 1997.

[Ruh 1997] Ruh, William A., and Mowbray, Thomas J., *Inside CORBA*, Reading, MA: Addison - Wesley, 1997.

[Schmidt 1995] Schmidt, Douglas, "Using Design Patterns to Develop Reusable Object-Oriented Communication Software," *Communications of the ACM*, October 1995, pp 65–74.

[Schmidt 1995] Schmidt, Douglas C., and Coplien, James O., *Pattern Languages of Program Design*, Reading, MA: Addison-Wesley, 1995.

[Shaw 1993] Shaw, M. "Software Architecture for Shared Information Systems," Carnegie Mellon University, Software Engineering Institute, Technical Report No. CMU/SEI-93-TR-3, ESC-TR-93-180, March 1993.

[Shaw 1996] Shaw, Mary, and Garlan, David, *Software Architecture: Perspectives on an Emerging*

*Discipline*, Englewood Cliffs, NJ: Prentice-Hall, 1996.

[Spewak 1992] Spewak, S.H., and Hill, S.C., *Enterprise Architecture Planning*, New York: John Wiley & Sons, Inc., 1992.

[Strikeleather 1996] J. Strikeleather, "The Importance of Architecture," *OBJECT* 6(2), April 1996.

[Taylor 1992] Taylor, D.A., *Object-Oriented Information Systems*, New York: John Wiley & Sons, Inc., 1992.

[Vlissides 1996] Vlissides, John M.; Coplien, James O.; and Kerth, Norman L., *Pattern Languages of Program Design*, Reading, MA: Addison-Wesley, 1996.

[Walden 1995] Walden, Kim, and Nerson, Jean-Marc, *Seamless Object-Oriented Software Architecture*, Englewood Cliffs, NJ: Prentice-Hall, 1995.

[Webster 1995] Webster, Bruce F., *Pitfalls of Object-Oriented Development*, New York: M&T Books, 1995.

[Webster 1997] Webster, Bruce F., "Everything You Know Is Wrong," Object World West '97, SOFTBANK-COMDEX, 1997.

[Yourdon 1993] Yourdon, Edward, Software Reusability: *The Decline and Fall of the American Programmer*, Englewood Cliffs, NJ: Prentice-Hall, 1993.

[Yourdon 1997] Yourdon, Edward, *Death March*, Short Hills, NJ: Prentice-Hall, 1997.

# 索　引

索引中的页码为英文原书的页码，与书中边栏的页码一致。

## B

## C

## E

## F



## G

## H



## I

## J



## K



## L

## M

## R

## S

## T



## U



## V

## W



## Z